

Cromemco 32 K Structured Basic



This manual was produced on a Cromemco System Three computer using the SCREEN Editor. The edited text was formatted using the Cromemco Word Processing System Formatter. Final camera-ready copy was printed on a Cromemco 3355 printer.

TABLE OF CONTENTS

1.	Introduction	11
2.	Instruction Syntax	15
2.1	Spaces or Blank Characters	15
2.2	Upper Case Characters	15
2.3	The Basic Prompt	16
2.4	Commands	16
2.5	Statements	16
2.5.1	Line Names	17
2.5.2	Multiple Instruction Lines	18
3.	Numeric and String Internal Machine Representation . .	21
3.1	Integer	21
3.2	Short Floating Point	22
3.3	Long Floating Point	23
3.4	Hexadecimal	23
3.5	String	23
4.	Constant and String Literal Formats	25
4.1	Integer and Floating Point Constants	25
4.1.1	Storage of Floating Point Constants . .	26
4.2	Hexadecimal Constants	26
4.3	String Literals	27
5.	Variable Representation	29
5.1	Numeric Variables	29
5.1.1	Format	29
5.1.2	Integer Variables	30
5.1.3	Short Floating Point Variables	30
5.1.4	Long Floating Point Variables	30
5.1.5	Matrices and Lists	30
5.1.5.1	Format	31
5.2	String Variables	32
5.2.1	Format	32
5.2.2	Dimensioning String Variables	32

5.2.3	Referencing String Variables	33
5.2.3.1	Format-1	34
5.2.3.2	Format-2	34
5.2.3.3	Format-3	35
5.2.3.4	Format-4	36
6.	Operators	39
6.1	Arithmetic Operators	39
6.2	Assignment Operator	40
6.3	Relational Operators	42
6.4	Boolean Operators	45
7.	Programming Examples	49
7.1	Getting Started	49
7.1.1	The Command or Immediate Mode	50
7.1.2	The RUN or Program Execution Mode	54
7.1.3	Activating the System Printer	56
7.1.4	Program Editing	57
7.2	Example Program	60
7.2.1	Listing to a Disk File	61
7.2.2	Entering From a Disk File	62
7.2.3	Using the Save and Load commands	63
7.3	Statistical Analysis Program	64
8.	Program Development	67
	Automatic Line Numbering	69
	Bye	70
	Delete Statement Lines	71
	Directory	73
	Edit	75
	Find	76
	Change	77
	Enter File	78
	List Current Program	79
	List Variables	81
	Load Program	83
	Renumber Statement Lines	84
	Run Program	88
	Save Program	90
	Scratch User Area	91
	Enable Trace Option	93
	Disable Trace Option	94

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL

9.	Documentation	95
	Remark	95
10.	Assignment Instructions	97
	Let	97
	Matrix Initialization	99
11.	Initialization	101
	Degree Mode	101
	Dimension	102
	Integer Mode	104
	Integer Variable	105
	Long Floating Point Mode	106
	Long Variable	108
	Radian Mode	109
	Short Floating Point Mode	110
	Short Variable	111
12.	Control Structures	113
	Continue Program Execution	113
	End Program Execution	114
	For-Next Loop	115
	Gosub-Return	119
	Gosub-Retry	121
	Goto	123
	If-Then	125
	If-Then-Else	127
	On-Goto	129
	On-Gosub	129
	Repeat-Until Loop	131
	While-Endwhile Loop	132
	Stop Program Execution	135
13.	Console and Data Input/Output	137
	Input (from the console)	137
	Print (to the console)	140
	Read Data	143
	Restore Data Pointer	145
	Data	147

14.	Output Formatting	149
	Print Using	149
	Space	157
	Tab	158
15.	Data File Input/Output	161
	Discussion Section	161
	15.1 Data Files	161
	15.1.1 Records	161
	15.1.2 Fields	162
	15.2 Floppy Diskette	162
	15.3 Creating a Data File	162
	15.3.1 Opening an I/O Channel	163
	15.3.2 Close	163
	15.4 Internal Machine vs. ASCII Representation	163
	15.5 Print and Input	164
	15.6 Put and Get	165
	15.7 An Interesting Note	165
	15.8 File Pointer	166
	15.9 Sequential Files	167
	15.10 Random Files	168
	15.11 CDOS DUMP Utility	168
	Instruction Section	169
	Create File	169
	Open File	170
	Close File	172
	Erase File	173
	Rename File	174
	Print	175
	Input	178
	Put Record	180
	Get Record	182
16.	Functions	187
	16.1 Arithmetic Functions	190
	Absolute Value	190
	Binary Operations	191
	Exponent	193
	Fractional Portion	194
	Integer Portion	195
	Integer Random Number Generator	196
	Logarithm	197
	Maximum Value	198
	Minimum Value	199
	Randomize	200

	Random Number Generator	201
	Sign	202
	Square Root	203
16.2	Trigonometric Functions	204
	Arctangent	204
	Cosine	205
	Sine	206
	Tangent	207
16.3	Programmer Defined Functions	208
16.4	String Functions	211
	ASCII Value of a Character	211
	Character	212
	Expand String	213
	ASCII Hex Representation	214
	Length of String	215
	Position of Substring	216
	String Equivalent	218
	Value of String	219
	Value of String With Error Checking	220
17.	System and File Status	223
	Disk Drive	223
	Enable Echo	224
	Disable Echo	225
	Enable Escape	226
	Disable Escape	227
	Free Space	228
	I/O Status	229
	On Error Transfer Control	231
	On Escape Transfer Control	233
	Set System Parameter	235
	System Parameter	237
18.	Machine Level Instructions and Functions	239
	Address of a Variable	239
	Input From I/O Port	240
	Output To I/O Port	242
	Peek At Memory	243
	Poke Into Memory	244
	Call a User Program	245
	Type of Variable	248
	Basic-KSAM Numeric Sorting Conversions	249

19.	Scope of Variables	251
	Common Storage Area Method I	251
	Common Storage Area Method II	253
	Define Local Variable	256
20.	Procedures	261
	Library Builder	263
	Procedure Call	265
	Procedure Definition	269
	Procedure End	270
	Procedure Error End	271
	Procedure Exit	272
	Clear Partition	273
	Select Procedure Library	274
	Use Partition	275
	Lock Partition	276
	Unlock Partition	277
21.	Program Security	279
	Delete Remark Statements	279
	Protect Program Lines	281
22.	Basic-KSAM	283
	Discussion Section	283
	22.2 Structure of Basic-KSAM Files	285
	22.3 Logical Records and Keys	287
	22.4 Alternate Key Files	290
	22.5 The Current Record Pointer	294
	Summary of Basic-KSAM Instructions	296
	File Instructions	301
	Create Primary Data File	302
	Close File	304
	Open Primary File	305
	Add Volume to Existing File	307
	Sequential Access Instructions	308
	Read Previous Record, Primary File	309
	Read Current Record, Primary File	310
	Read Next Record, Primary File	311
	Retrieve Primary Key, Current Record	312

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL

Random Access Instructions	313
Read Random Record, Primary File	315
Read Approximate, Primary File	316
Update Record, Primary File	318
Delete Record, Primary File	319
Read Nth Record, Primary File	320
Add Record, Primary File	321
Load Record, Primary File	323
Alternate Key Instructions	324
Create Alternate Key File	325
Open Alternate File	327
Read Primary Record By Current Alternate Key	329
Read First Primary Record By Specified Alternate Key	331
Read Next Primary Record By Current Alternate Key	332
Verify Alternate Record	334
Add Record, Alternate File	335
Delete Record, Alternate File	336
Example Program	337
22.7 Basic-KSAM Error Messages	340
Summary	340
Explanation	341
22.8 Basic-KSAM Utilities	346
Summary	346
Explanation	349
23. Glossary	353
24. Error Messages	369
24.1 Fatal	369
24.2 User Trappable (Non-Fatal)	374
25. Appendix	381
ASCII Character Codes	381
Overlay Structure	382
Automatic Startup And Program Execution From CDOS	383
Mixed Mode Arithmetic	385
Patch Space	388
Areas of User Interest	389
Adding Device Drivers To Basic	392
DDLIST (Device Driver List)	398
Accessory I/O Drivers	401
Changing the Number of I/O Channels	403

1. Introduction

An Overview of the Design Philosophy of Cromemco 32K Structured Basic

Introduction

There are many powerful software tools currently available to the computer programmer. These tools are in the form of low level languages, high level languages, versatile operating systems, text editors, debugging facilities, and linkable standard modules.

During the past decade there has been an increasing awareness of the difficulties in developing complex computer programs. A trend has been observed concerning the nature of complex program development. This trend can be stated simply as follows:

1. Programmer productivity decreases with increasing program size. Program size is typically expressed in terms of a number of lines of code.
2. Productivity falls dramatically for programs of greater than 1000 lines of program code.
3. Modular programming practices increase programming productivity in large programming tasks.

A number of languages exist which facilitate modular programming practices. These languages are called structured languages. A structured language allows the programmer to create blocks of code which comprise the program. Structured languages also have program control statements that allow and insure that these blocks of code can operate independently of other blocks of code. Thus standard modules can be written that might consist of many of these blocks.

Cromemco 32K Structured Basic is designed to allow these important programming practices to be used in the very popular Basic programming language that

1. Introduction

originated at Dartmouth College more than fifteen years ago. In addition, it is designed to provide the programmer with the most computing power per line of code ever offered in a version of Basic. This increase in programming power is provided to enable the programmer to quickly write programs that do more than ever before and to assist the programmer in writing programs that are able to work correctly on the first try.

32K Structured Basic for Quick Software Development

It is in this emphasis on quick programs, programs of less than 10,000 lines of code, that Cromemco 32K Structured Basic differs from other structured languages, such as Algol and Pascal. These other languages restrict the programming practices to those appropriate for major software undertakings that might be expected to take several years of programming and involve many programmers working together and creating tens of thousands of lines of program code. The interactive power of Basic and the philosophy of its versatility and error tolerance are intentionally not implemented to discourage poor practices for large programs. Basic is thought to be a friendly language and the language of choice for small and medium-sized jobs by many programmers. This friendliness of the language comes from there being very few restrictions placed on the programmer thus making the language easy to learn. For instance, program execution may be stopped using the Escape key on the programmer's terminal, the present value of program variables may be displayed and changed, followed by changes in a few program statements, and execution can be commanded to continue with these changes immediately incorporated.

Cromemco 32K Structured Basic extends the capability of the popular and powerful Cromemco 16K Extended Basic. These extensions are provided to enable programs to be written more quickly and make it more likely that these programs will work on the first try. These extensions include the following new features:

1. Variable names up to 31 characters, such as "Present'worth" and "Interest'rate" may be used. All characters in the name are verified

1. Introduction

to insure uniqueness.

2. Statement labels may be referenced in addition to line numbers. "Goto Io'driver" is a valid instruction.
3. REPEAT, WHILE, IF-THEN-ELSE, and PROCEDURE statements of structured languages are provided.
4. Module handling instructions may be used to create programs that are swapped into memory from a Procedure Library file on disk. This automatic loading of software modules, each with its own local variables, makes it easy to write programs of indefinite length without regard for the computer Random Access Memory capacity.
5. Extended string functions allow the insertion of strings within strings and unambiguous conversion of strings to numeric representations.
6. A Keyed Sequential Access Method (KSAM) is provided to simplify and speed up contents-oriented file accesses such as are found in data base programming.
7. A line-oriented text editor is provided to facilitate programming changes.
8. The LVAR command is provided to enable all the variables used in the program to be listed.

32K Structured Basic: Built on a Powerful Foundation

Cromemco 32K Structured Basic is an extension of the popular and capable Cromemco 16K Extended Basic. Cromemco 16K Extended Basic is popular among microcomputer programmers. Among its most outstanding features are:

1. High execution speed resulting from a semicompiling design. Execution rate is as fast for large programs as it is for small programs. This feature is not available in most other versions of Basic.

1. Introduction

2. High accuracy 14 digit arithmetic. The binary coded decimal arithmetic used in 16K Extended Basic is essential for accounting and business programming applications.
3. Extensive string handling and output formatting capabilities. These Cobol-like capabilities make report and forms generation easy.
4. Versatile random and sequential disk access methods.
5. Automatically indented listings denoting loops and other control structures.

2. Instruction Syntax

INSTRUCTION SYNTAX

The Cromemco Structured Basic language is designed to allow the user to structure and format a program in a wide variety of styles. This section covers the features which allow this flexibility as well as those elements of Basic syntax which do affect program operation.

2.1 Spaces or Blank Characters

In Cromemco 16K Extended Basic, spaces within an instruction were generally not significant. The following two instructions were equivalent.

```
GOTO10  
GOTO 10
```

However, in 32K Structured Basic, GOTO10 is a legal variable name and not an instruction at all. To make an instruction, the space after the key word GOTO is essential.

In most cases, a space is now required after a Basic key word.

2.2 Upper Case Characters

Basic instructions can be entered in any combination of both upper and lower case characters. Upon LISTing a program, Basic will convert key words and variable names into its own format, initial character uppercase, the rest lower case. REMarks and string literals will remain exactly as entered.

2. Instruction Syntax

2.3 The Basic Prompt (>>)

When Basic is ready to accept a command or statement line, it displays a prompt which consists of two greater than symbols (>>). The purpose of the prompt is to indicate that Basic has finished its last task and is waiting for the user's next instruction.

2.4 Commands

A command is a Basic instruction which is executed immediately (as soon as the carriage RETURN key is depressed). Commands have no line numbers because they are not stored by Basic.

Cromemco Structured Basic allows most instructions to be used as commands. For example, Basic can be used as a calculator while it is in the command mode:

```
>>Print 20000/5  
4000
```

```
>>Print (4000+77)/63.  
64.714285714285
```

As can be seen, a command may be given whenever the Basic prompt (>>) is displayed.

In this manual, the term instruction includes both commands and statements. All instructions listed as instructions or commands may be used as commands, while those listed as statements may not be used as commands.

2.5 Statements

A statement is a Basic program line which contains one or more instructions, and which is stored for execution at a later time. Statements are not executed until the RUN command, or some other command which will begin execution of a program, is given. A statement line has a unique line number (within a program or Partition) by which it can be accessed. If a second statement is entered with the same line number as a line which already

2. Instruction Syntax

exists, the original line will be replaced by the new line.

Cromemco Structured Basic allows most instructions to be used as statements. For example, one Basic program can LOAD and RUN another Basic program:

```
100 Run "Prog2"
```

Execution of this statement will cause Prog2 to be LOAded into the User Area and execution to begin.

In this manual, the term instruction includes both statements and commands. All instructions listed as instructions or statements may be used as statements, while those listed as commands may not be used as statements.

2.5.1 Line Names

A statement line may be referenced either by a line number or by an alphanumeric line name. A line name includes from 1 to 31 characters. The first character must be alphabetic. Each of the remaining characters may be alphabetic, numeric, or the apostrophe (').

A line is labeled with a line name by following the line number with an asterisk and line name. The asterisk and name is considered an instruction which declares the name of the line. The line may contain additional instructions separated by colons as specified by Basic syntax.

Label names are used with GOTO and GOSUB to transfer control from one part of a program to another. They may also be used to reference various parts of a program for EDITing, statement RENUMBERing, etc.

2. Instruction Syntax

Example:

```

10 *Beginning
.
.
50 Gosub Get'data
60 Gosub Process'19
.
.
90 Goto Beginning
.
.
300 *Get'data : Data'pointer=1
.
.
360 Return
400 *Process'19 : Error'flag=0
.
.
450 Return

```

2.5.2 Multiple Instruction Lines

Cromemco Structured Basic allows more than one instruction to be associated with a single line number. Each pair of adjacent instructions must be separated by a colon (:). The number of instructions which may appear on a single line is limited only by the length of a line. For example:

```

10 *Start : Input Quan : If Quan<0 Then Goto Start
20   Print Sqr(Quan) : Print : Print
30   Rem Line 20 will print, then skip 2 lines

```

User defined functions (DEF FNs) and DATA statements must appear as a single instruction on a line. The following instructions may appear as part of a multi-instruction line but they must be the last instruction on the line, i.e., no other instruction may follow on the same line.

2. Instruction Syntax

DELETE
ENDPROC
ENTER
ERRPROC
EXITPROC
LIST
ON-GOSUB
ON-GOTO
REM
RUN
SCR

A colon may not terminate a multiple instruction line. A colon must be followed by another instruction.

The IF-THEN instruction is unique when followed by other instructions on the same line. The reader is referred to the IF-THEN instruction for a further discussion.

NUMERIC AND STRING
INTERNAL MACHINE REPRESENTATION

Numeric and string alphanumeric information is stored by Basic in different forms. This section explains these different formats.

3.1 Integer

Integers are whole numbers and in Structured Basic they must be within the range +32767 to -32768. When stored by Basic, an integer occupies 2 bytes of memory.

Integer numbers are stored low byte first, high byte second. If the high bit of the high byte is a 1, then the number is negative; if it is a 0, then the number is positive.

A positive number is stored as the binary representation of the number.

A negative number is stored as the 2's complement of the number.

Example:

The number 1234 will be represented by the hexadecimal bytes D2 04 when stored as an Integer. When the order of the bytes is reversed (04 D2) this is the binary equivalent of the decimal number 1234.

The number -1234 will be represented by the hexadecimal bytes 2E FB when stored as an Integer. When the order of the bytes is reversed (FB 2E) this is the 2's complement of the binary equivalent of the decimal number 1234. Because the high bit of the high byte was 1 (before the 2's complement was taken), the number is negative (-1234).

3. Numeric and String Internal Machine Representation

3.2 Short Floating Point

A Short Floating Point number stored by Basic occupies 4 bytes, has an accuracy of 6 digits, and must be within the range $\pm 9.99\text{E}+62$ to $\pm 9.99\text{E}-65$.

The first byte of a Short Floating Point number contains the sign of the number and the exponent in excess 40H (64 decimal) notation. If the high bit of the first byte is a 1, then the number is negative; if it is a 0, then the number is positive. Note that this is not the sign of the exponent but rather of the number itself. The remaining 7 bits of the first byte contain the exponent plus 40H. In order to find the true exponent, 40H (64 decimal) must be subtracted from this number.

The remaining three bytes contain the BCD (Binary Coded Decimal) mantissa which has been normalized to a value between 0 and 1. The implicit decimal point is located before the first byte of the mantissa. Each byte of the mantissa can contain 2 significant digits yielding a total of 6 significant digits for a Short Floating Point number.

Example:

The number $-1.2345\text{E}+21$ will be represented by the hexadecimal bytes D6 12 34 50 when stored as a Short Floating Point number.

The first bit of the first byte is a 1 indicating that the number is negative. The remaining 7 bits of the first byte (56H or 86 decimal) is the exponent plus 40H. To get the true exponent 40H must be subtracted from 56H. This leaves an exponent of 16H or 22 decimal. This is not the exponent of the original number because the number was normalized. In this example, normalization involved dividing the number by 10 and adding 1 to the exponent to compensate for the division.

The remaining three bytes are the BCD representation of the normalized number. BCD stands for Binary Coded Decimal which is a method of representing a decimal number in binary. Using this method, each byte can contain two one digit

3. Numeric and String Internal Machine Representation

decimal numbers. As can be seen from the example, non-significant digits are zero filled.

3.3 Long Floating Point

A Long Floating Point number stored by Basic occupies 8 bytes, has an accuracy of 14 digits, and must be within the range $\pm 9.99\text{E}+62$ to $\pm 9.99\text{E}-65$.

The internal representation of a Long Floating Point number is similar to that of a Short Floating Point number. The difference is that four additional bytes are added to the mantissa for a total of 7 bytes. This is how a Long Floating Point number can maintain 14 significant digits.

3.4 Hexadecimal

A Hexadecimal number occupies two bytes and must be within the range 0H to FFFFH.

The internal representation of a hexadecimal number is the same as that of an Integer. Except for the binary functions, hexadecimal numbers are treated as signed integers (refer to section 3.1.)

3.5 String

A string is an ordered list of alphanumeric information. Examples of strings include words, sentences, parts of words, groups of letters or special characters.

Elements of a string (characters) stored by Basic occupy 1 byte each. They are represented internally as an eight bit number which is normally considered to be the ASCII (American Standard Code for Information Interchange) code for the character being stored. Bit number 7 (the high bit) is a parity bit in the ASCII convention (it is normally zero in Structured Basic) and, although it effects string comparisons, it does not effect the character which is PRINTed. Only GET and CHR\$ can return a string byte with the high bit set (=1).

4. Constant and String Literal Formats

CONSTANT AND STRING LITERAL FORMATS

Constants are, as the name implies, unchanging. They each have one value.

String literals are similar to constants in that they each maintain one value which does not change.

This section covers the standard formats for constants and string literals in Structured Basic.

4.1 Integer and Floating Point Constants

A constant is a number. It does not change value and is represented as it would be in any arithmetic computation.

There are three types of constants: integer, floating point, and hexadecimal. All constants (not specified as hexadecimal) equal to or greater than 10,000 and those containing a decimal point are always stored by Basic as floating point numbers (either Short or Long Floating Point depending on the current mode, but always Long if there are more than 6 significant digits). All constants (not specified as hexadecimal) with a value less than 10,000 and not containing a decimal point are stored by Basic as Integer numbers.

Constants

<u>Floating Point</u>	<u>Integer</u>
20000	55
3.	9985
.000376	5
12.7	458

4. Constant and String Literal Formats

4.1.1 Storage of Floating Point Constants

Floating point constants (1.2, 3., 1E6, etc.) are stored according to the mode which is active when they are entered. An exception is a constant with more than 6 significant digits, which is always stored as a Long Floating Point number. For example, if the active mode is set to Short Floating Point (see the SFMODE instruction) and the following commands are given:

```
>>Long Long'num
>>Long'num = 1./3.
```

Long'num will be assigned a value of 0.33333300000000 because 1. and 3. are SHORT (SFMODE was current when they were entered). But, under the same circumstances, if we had said:

```
>>Long Long'num
>>Long'num = 1./3.000000000
```

Long'num would have been assigned a value of 0.33333333333333 because 3.000000000 is forced to LONG (more than 6 significant digits).

If in the two previous examples, the current mode had been Long Floating Point, Long'num would have received a value of 0.33333333333333 in both cases because both 1. and 3. would have been stored as Long Floating Point numbers.

4.2 Hexadecimal Constants

Hexadecimal numbers are used in base 16 arithmetic. The set of digits used in hexadecimal arithmetic is 0 through 9 and A through F.

In Cromemco Structured Basic, hexadecimal constants are identified by leading and trailing percent (%) signs. The trailing percent sign may be omitted when this will not create an ambiguous expression.

4. Constant and String Literal Formats

Constants

<u>Hexadecimal</u>	<u>Equivalent Decimal</u>
%8000%	-32768
%9000%	-28672
%FFFF%	-1
%9%	9
%A%	10
%F%	15
%10%	16
%80%	128
%FF%	255
%100%	256
%7FFF%	32767

A hexadecimal number is stored in the same format as an Integer and may be used wherever a constant is allowed.

4.3 String Literals

A string literal is a string which is enclosed between quotation marks. The quotation marks are not part of the string itself, but are used to delimit (mark the ends of) the string. The value of a string literal does not change.

A quotation mark can be represented within a string literal by the use of two quotation marks, one immediately following the other.

String Literals

```
"This is a string literal"
"Here are imbedded ""quotation"" marks"
"{special} *+- characters are o.k.!"
"-----"
"                                just spaces"
```

The trailing quotation mark on a string literal is not required at the end of a line. In this case, the carriage RETURN will terminate, but not be a part of, the string literal.

5. Variable Representation

VARIABLE REPRESENTATION

Variables are, as the name implies, able to change or vary in value. As a program is executed, variables may be assigned new values at any time. This section covers the standard formats for variables in Structured Basic.

5.1 Numeric Variables

Numeric variables may be assigned numeric values. The range and accuracy of these variables depends on their type. Refer to the Chapter on Numeric Internal Machine Representation for more information.

5.1.1 Format of Numeric Variables

A variable name includes from 1 to 31 characters. The first character must be alphabetic. Each of the remaining characters may be alphabetic, numeric, or the apostrophe (').

Variable names may be entered in lower or upper case characters, or any combination of the two. Basic will convert the variable name into its format which is an initial upper case character followed by all lower case characters.

The following are examples of legal variable names:

```
Interest'rate
A0
A123
Name1$
Price'to'earnings'ratio
Stock'number
```

5. Variable Representation

5.1.2 Integer Variables

Specific variables may be set to INTEGER by the INTEGER instruction. The default mode for variables may be set to INTEGER if the IMODE instruction is given before the RUN instruction. Refer to the sections covering Integer Internal Machine Representation, and the INTEGER and IMODE instructions.

5.1.3 Short Floating Point Variables

Specific variables may be set to Short Floating Point by the SHORT instruction. The default mode for variables may be set to Short Floating Point if the SFMODE instruction is given before the RUN instruction. Refer to the sections covering Short Floating Point Internal Machine Representation, and the SHORT and SFMODE instructions.

5.1.4 Long Floating Point Variables

Specific variables may be set to Long Floating Point by the LONG instruction. Long Floating Point mode is normally the default mode. If the mode has been changed to SFMODE or IMODE, the default mode for variables may be reset to Long Floating Point if the LFMODE instruction is given before the RUN instruction. Refer to the sections covering Long Floating Point Internal Machine Representation, and the LONG and LFMODE instructions.

5.1.5 Matrices and Lists

A matrix is an array of numeric variables in a prescribed form. For example, the array:

3	2	0
1	4	6
-3	4	5

is a matrix with three rows and three columns. A matrix with m rows and n columns is written:

5. Variable Representation

```

a11 a12 a13 ...a1n
a21 a22 a23 ...a2n
.
.
.
am1 am2 am3 ...amn

```

The individual entries in the matrix are called elements or cells. For example the quantity a_{ij} in the above matrix is the element in row i and column j . Subscripts used to indicate elements always denote the row first and the column second. Cromemco Basic permits the user to define one, two, or three dimensional matrices. A two (i.e., M_{ij}) or three (i.e., M_{ijk}) dimensional matrix is commonly called a table. A one dimensional matrix, a matrix with n columns but only one row, is commonly called a list. For example, the matrix:

```
3, -1, 5, -8
```

is a list (or a matrix) with one row and four columns.

A matrix may be defined to be composed of Long or Short Floating Point or Integer variables.

5.1.5.1 Format

A matrix is named in the same manner as a numeric variable. A specific element of a matrix is accessed by the matrix name followed by 1, 2, or 3 indices enclosed in parentheses. These indices may be numeric expressions, variables, or constants.

Singly DIMensioned matrices are implicitly DIMensioned as 10. Larger singly DIMensioned as well as 2 and 3 DIMensional matrices must be explicitly DIMensioned by the DIMension, INTEGER, LONG, or SHORT instruction.

An element of a matrix may be used anywhere a numeric variable is allowed.

5. Variable Representation

5.2 String Variables

String variables may be assigned alphanumeric values. This includes all letters (both upper and lower case), numbers, and all printable and non-printable characters. Refer to the section on String Internal Machine Representation for more information.

5.2.1 Format

A string variable name includes from 1 to 31 characters. The first character must be alphabetic. Each of the remaining characters may be alphabetic, numeric, or the apostrophe ('). The last character must be a dollar sign (\$).

String variable names may be entered in lower or upper case characters, or any combination of the two. Basic will convert the string variable name into its format which is an initial upper case character followed by all lower case characters.

The following are examples of legal string variable names:

```
Text$  
Page$(Character)  
Last'name$  
Address$
```

5.2.2 Dimensioning String Variables

There is, for most purposes, no limit to the size (i.e., number of characters) of a string literal that may be assigned to a string variable. However, the default value in Structured Basic for string size is 11 or fewer characters. If string values of more than 11 characters are to be assigned to a variable, the string variable must be DIMensioned. The DIM statement is used in Basic to define the size of a string variable (see the DIM instruction).

5. Variable Representation

Example:

```
10 Dim Apples$(20), Pears$(30), Bananas$(40)
```

In this example, the string variable Apples\$ is DIMensioned to allow for strings up to 21 characters in length, the variable Pears\$ is dimensioned to allow up to 31 characters, and Bananas\$ is dimensioned for up to 41 characters. Any string value assigned to a variable which exceeds the specified dimension is truncated. Consequently, the programmer should be sure to DIMension string variables to handle the largest string which will be input.

Note: DIM Apples\$(20) allows a 21 character string because string bytes are numbered from 0 through the specified DIM size. Remember that using the 0'th element of strings (and arrays) can save memory space.

A string of LENGTH zero is called a null string.

5.2.3 Referencing String Variables

String variables may be referenced (for input, output, manipulation, or comparison) in their entirety or by parts (substrings). For example, if the string variable Name\$ equals "Substring Example", substrings of Name\$ include "Sub", "Ubstr", "G Exa", and any other part of the string.

Substrings are referenced by subscripting string variables. The four formats for referencing string variables are discussed here. In these formats svar refers to a string variable name (refer to section 5.2.1) and aexp-1 and aexp-2 are arithmetic expressions, variables, or constants. If the string variable has not been explicitly DIMensioned, it is considered to have an implicit DIMension of 10.

5. Variable Representation

5.2.3.1 Format-1: svar

On input (INPUT or GET), if a string variable is referenced without subscripts, the entire string is referenced by implication. This means that all characters, 0 through the DIMension of the string are referenced. On output (PRINT or PUT), if a string variable is referenced without subscripts, the string is referenced from the first character (character number 0) through the last non-null character. This is the same as saying that the string is referenced from character number 0 for a length LEN (see the LEN string function).

Summary -

On input, if no subscripts are used, the entire string variable is referenced as DIMensioned.

On output, the string is referenced from character 0 through the last non-null character.

5.2.3.2 Format-2: svar(aexp-1)

If a single subscript is used it must be greater than zero and less than or equal to the DIMension of the string variable. This format defines a substring which starts with the character in position aexp-1 and includes all subsequent characters in the string through the DIMensioned value (input) or through the last non-null character (output). If, on output, a character position greater than the LENGth of the string is explicitly referenced, then all subsequent null characters are referenced by implication (Refer to the LENGth string function).

If aexp-1 is less than zero, the entire string will be referenced (character 0 through the DIMensioned value).

If aexp-1 is greater than the DIMension of the string variable, a run time error message will result.

5. Variable Representation

Summary -

If one (aexp-1) subscript is used, the substring referenced starts with character in position aexp-1 and includes all subsequent characters in the string through the DIMension value (input) or through the last non-null character (output).

$0 < \text{aexp-1} \leq \text{DIMension of svar}$

5.2.3.3 Format-3: svar(aexp-1,aexp-2) aexp-2 ≥ 0

In this format, aexp-1 must be greater than or equal to zero and less than the DIMension of the string variable. Thus, aexp-1 defines the starting character of the substring as it did in format-2.

The second subscript (aexp-2) must be greater than the first subscript (aexp-1) and less than or equal to the DIMension of the string variable. Now, aexp-2 defines the final character of the substring.

If either aexp-1 or aexp-2 exceeds the DIM value of the string variable, a run time error will result.

If aexp-1 is less than zero, the entire string is referenced as though no subscript had been used (see format-1).

If aexp-2 is less than aexp-1, aexp-1 will be the starting character of the substring and the DIMensioned value will be the final character (as though format-2 had been used with aexp-1).

Refer to format-4 if aexp-2 is less than zero.

Summary -

If two subscripts (aexp-1, aexp-2) are used, the second subscript being positive, the substring referenced starts with character in position aexp-1 and ends with character in position aexp-2.

$0 \leq \text{aexp-1} < \text{aexp-2} < \text{DIMension of svar}$

5. Variable Representation

5.2.3.4 Format-4: `svar(aexp-1,aexp-2)` `aexp-2 < 0`

In this format, `aexp-1` must be greater than or equal to zero and less than the DIMension of the string variable. Thus, `aexp-1` defines the starting character of the substring as it did in format-2 and format-3.

The second subscript (`aexp-2`) must be less than zero and its absolute (positive) value must, when added to `aexp-1` not exceed the DIMensioned value of the string variable. In this case, the absolute (positive) value of `aexp-2` indicates the length or number of characters included in the substring.

If `aexp-1` plus the absolute value of `aexp-2` minus one is greater than the DIMensioned value of the string variable, a run time error will result.

If `aexp-1` is less than zero, the entire string is referenced as though no subscripting had been used (see format-1).

Refer to format-3 if `aexp-2` is equal to or greater than zero.

Summary -

If two subscripts (`aexp-1`, `aexp-2`) are used, the second subscript being negative, the substring referenced starts with character `aexp-1` and continues for a length of `|aexp-2|`.

$0 \leq \text{aexp-1} < \text{DIMension of svar}$
 $\text{aexp-1} + |\text{aexp-2}| \leq \text{DIMension of svar}.$

Note:

If `aexp-2` is equal to `%8000%` then `aexp-2` is considered to be equal to `-0` (minus zero) and will produce a string of length zero (i.e., a null string).

Examples:

Assume that the command `Wd$ = "ABCDEFGHIIJK"` has been given and that the string variable `Wd$` has not been DIMensioned in a DIMension instruction (it is implicitly DIMensioned at 10 and it contains 11 characters numbered zero through ten).

5. Variable Representation

<u>Instruction</u>	<u>Result</u>	<u>Explanation</u>
Print Wd\$	ABCDEFGH IJK	Format-1 was used to PRINT the entire string.
Print Wd\$(5)	FGHIJK	Format-2 was used to PRINT the substring starting with character 5 (remember that the characters are numbered starting with 0).
Print Wd\$(4,8)	EFGHI	Format-3 was used to PRINT the substring starting with character 4 and ending with character 8.
Print Wd\$(3,-2)	DE	Format-4 was used to PRINT the substring starting with character 3, for a length of two characters.

Now assume the command Wd\$ = "ABCDEFG" has been given. This will fill the string with the seven characters A-G and 4 null characters. In the following examples the lower case n represents a null character.

<u>Instruction</u>	<u>Result</u>	<u>Explanation</u>
Print Wd\$	ABCDEFG	Format-1 was used to PRINT character 0 through the first non-null character.
Print Wd\$(-1)	ABCDEFGnnnn	Format-2 (with a negative argument) was used to PRINT characters 0 through Dim.
Print Wd\$(8)	nnn	Format-1 was used to PRINT characters 8 through 10. Note that character 8 is greater than the LENGTH of Wd\$ so that all succeeding nulls are referenced by implication.
Print Wd\$(5)	FG	Format-2 was used to PRINT character 5 through the last non-null character.

5. Variable Representation

Print Wd\$(5,2) FGnnnn

Format-3 was used with
aexp-2<aexp-1 so the string was
output from character 5 through
the DIMension of the string.

Print Wd\$(8,9) nn

Format-3 was used to output
characters 8 through 9.

Print Wd\$(13,15)

Format-3 error, aexp-1>Dim.

Print Wd\$(3,%8000%)

A length of -0 indicates a null
string, no string is output.

6. Operators

OPERATORS

An operator is a symbol or group of letters which indicate that an action is to be taken on one or two items. This section describes the types of operators used in Structured Basic and the symbols which represent them.

6.1 Arithmetic Operators

The arithmetic operators are analogous to their algebraic counterparts:

<u>Arithmetic Operator</u>	<u>Meaning</u>
+	plus sign (positive number)
-	minus sign (negative number)
** or ^	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

Arithmetic operations in a numeric expression are performed, according to the priority of the operation, from left to right. All operations enclosed within parentheses are performed first. When multiple sets of parentheses appear, the operations in the innermost set of parentheses are performed first, followed by the operations in the next set of parentheses, and so on until the operations in the outermost set of parentheses are evaluated. Following evaluation of expressions enclosed in parentheses, arithmetic operations are performed in the following order: plus and minus signs (unary operators), exponentiation, division and multiplication (these two operations have the same priority), and addition and subtraction (these

6. Operators

operations also have the same priority). When operations have the same priority, calculations are performed from left to right within the expression. The use of parentheses can alter the order in which operations are performed since the parentheses override both the left to right priority and the normal order of operations.

Examples:

A + B * C ** D

The above expression will be evaluated as follows:

```
temp = C ** D
templ = temp * B
final value = templ + A
```

The order of association can be changed by the use of parentheses:

((A + B) * C) ** D

This expression will be evaluated as follows:

```
temp = A + B
templ = temp * C
final value = templ ** D
```

It is a good idea to use parentheses if there is any doubt as to the order in which a series of operations will be performed. Intermediate results may also be assigned to temporary variables if this will help to clarify the order of operations.

6.2 Assignment Operator

The equal sign (=) is the assignment operator. It is used to assign the value of an arithmetic, relational, or Boolean expression, or a function, to a numeric variable.

In addition, the assignment operator is used to assign the value of a string literal, string variable, or string function to a string variable.

Basic also uses the equal sign as a relational operator (see the following section). The only place the equal sign is a legal assignment operator

6. Operators

is the first equal sign in a LET, implied LET, or MAT instruction.

Examples:

```

100  Number = 2
110  Print Number
120  Number = Number*2
130  If Number >= 16384 Then Stop
140  Goto 110

```

This program will print out all of the powers of 2 which are less than 16384 and then it will Stop. Statement 100 assigns the value of 2 to the numeric variable Number.

Line 120 assigns the result of the multiplication of the variable Number times the constant 2 to the variable Number. This is a good example of the reason the term assigned to is used instead of equal to. Number is obviously not equal to Number times 2. The expression on the right side of the assignment operator is evaluated (Number * 2) and then this value is assigned to the variable on the left of the assignment operator.

Statement 130 uses the greater than or equal to sign as a relational operator. The relational expression (Number >= 16384) is evaluated as true or false. If it is true, the portion of the instruction following Then is executed (program execution STOPS). If it is false, the next statement (140) is executed. Refer to the following section for a further discussion of relational operators.

Line 140 transfers control back to line 110 and execution continues until the relational expression (Number >= 16384) is true.

```

10  Termination'cond$ = "End"
20  *Loop : Print "Enter a name,"
30  Print "End to stop";
40  Input Name$
50  If Name$ = Termination'cond$ Then Stop
60  Print Name$ : Print
70  Goto Loop
80  End

```

Statement 10 assigns the value of the string literal "End" to the string variable

6. Operators

Termination'cond\$. Line 40 gets a string from the user and assigns the value of that string to the string variable Name\$ after having set Name\$ equal to a string of null characters (see the INPUT instruction).

Statement 50 uses the equal sign as a relational operator. The relational expression (Name\$ = Termination'cond\$ or we could have used Name\$ = "End") is evaluated as true or false and the execution of the program continues on the basis of the evaluation. See the next section for more information on relational operators.

6.3 Relational Operators

<u>Relational Operator</u>	<u>Meaning</u>
=	is equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to
<> or #	is not equal to

Relational operators are used to compare two expressions. Each of these expressions may be composed of other relational, Boolean, or arithmetic expressions. This allows the user to nest relational and Boolean expressions.

In addition, relational operators are used to compare string variables, string literals, and string functions with each other.

The result of a relational operation is either true (=1) or false (=0).

6. Operators

Examples:

```

100  If Test<=0 Then Goto 150
110  Print Test=0
120  Result=Test<0
130  Boolean=Test=5
150  End

```

Statement 100 can be read as, "If the value of the variable Test is less than or is equal to zero, transfer program control to line 150; otherwise continue with the next statement (line 110)." This is the most common use of a relational operator. This statement can be used to test the validity of user input, transferring control to another section of code if the input is not as desired.

Statement 110 may at first look like an improper assignment instruction. Remember that the equal sign is, in this case, a relational operator, not an assignment operator. This statement can be read as, "Compare the value of Test with 0 and PRINT the result of the comparison." Thus, a zero (=false) will be output if Test is not equal to zero and a 1 (=true) will be output if Test is equal to zero.

Statement 120 is similar to 110, with the exception that the result of the relational comparison is assigned to the variable Result. Result will therefore take on a value of 0 or 1.

Statement 130 is a combination of the ideas used in 110 and 120. Remember that the first equal sign is an assignment operator while the second equal sign is a relational operator. This statement can be read as, "Compare the value of the variable Test with five and assign the result of that comparison to the variable Boolean". Boolean will take on the value of 1 (=true) if Test is equal to five, otherwise Boolean will be set equal to 0 (=false).

```

300  Large$ = "zzz"
305  *Get'string : Input Test$
310  IF Test$ > Large$ Then Goto Get'string
315  Large$ = Test$
320  Print Large$
325  Goto Get'string

```

Line 300 initializes the string variable Large\$ by filling it with z's (the alphabetic character with the highest value in the ASCII collating sequence).

6. Operators

Line 305 requests string data from the user.

Statement 310 evaluates the relational expression (Get'string\$>Large\$) on the basis of the ASCII collating sequence (refer to the table of ASCII characters in the appendix). If the first character in the string Get'string\$ is different from the first character in the string Large\$ then these characters are compared and this relationship determines the relationship of the two strings. If the first character in each string is identical to the other, the comparison moves on the second character and so on. For example:

<u>Value of Get'string\$</u>	<u>Value of Large\$</u>	<u>Value of Expression Get'string\$>Large\$</u>
George	Fred	false
Fred	April	true
April	Ted	false
april	Ted	true
april	ted	false
april	apron	false
april	apricot	true
april	aprilandmay	false

Notice that all lower case letters follow the set of upper case letters in the ASCII collating sequence and that if two strings are the same except for length, the longer string has a greater value than the shorter one.

Continuing with the example, if the relational expression (line 310) is evaluated as true, control is transferred to line 305 and the user is asked for another string. If the expression is false, this means that the string which was just INPUT by the user is closer to the beginning of the ASCII collating sequence than any other string which had been INPUT previously. The value of this string is then assigned to the string variable Large\$ so that all strings which are INPUT following this one can be compared with it. The value of the string is then PRINTed and control is transferred to line 305. Execution of this program may be terminated by depressing the ESCape key.

6. Operators

6.4 Boolean Operators

<u>Boolean Operator</u>	<u>Meaning</u>
And	logical And
Or	logical Or
Xor	logical eXclusive Or
Not	logical Not or negation

Boolean operators perform a logical operation on one or two expressions. The expressions in a Boolean operation may take on one of two values: false (=0) or true (=1). In Cromemco Structured Basic, all values which are not equal to zero (false) are considered to be true when used with Boolean operators.

And Boolean Operator

The And Boolean operator compares two logical values and if both are 1 returns a result of 1. If both values are not 1, then the result is 0.

Truth Table

And	0	1
0	0	0
1	0	1

Or Boolean Operator

The Or Boolean operator compares two logical values and if either or both are equal to 1 then the result is 1. Otherwise the result is 0.

Truth Table

Or	0	1
0	0	1
1	1	1

6. Operators

Xor Boolean Operator

The Xor Boolean operator returns a 0 if the logical values are identical and a 1 if the logical values are not identical.

Truth Table

Xor	0	1
0	0	1
1	1	0

Not Boolean Operator

The Not Boolean operator returns the complement of any logical value. In other words, if the logical value is 1, the Not operator returns a 0. If the logical value is 0, a 1 is returned.

Not Truth Table

Not 0 is 1

Not 1 is 0

Examples:

```

10 True = 1 : False = 0
11 If True And True Then Print "11 True"
12 If Not True Then Print "12 True"
13 If Not False Then Print "13 True"
14 If True Xor True Then Print "14 True"
15 If Not (True Xor True) Then Print "15 True"
16 If True Or False Then Print "16 True"

```

The statements above give some examples of the use of Boolean operators. The IF-THEN instructions are used in these examples to test if a given Boolean expression is true or false. If the expression is true, the PRINT instruction following the IF-THEN is executed. If it is false, then nothing is printed.

The results of each of these expressions can be determined from the preceding truth tables. For example, line 14 uses the XOR operator to compare two true (=1) values. Looking at the XOR Truth Table, it can be seen that a 1 and a 1 yield a 0 or false value. Line 15 negates this same expression.

6. Operators

Looking at the Not Truth Table, NOT 0 (Not false) is seen to be equal to 1 or true.

The results of the rest of the examples can be determined in a similar manner, then tested on the computer.

7. Programming Examples

PROGRAMMING EXAMPLES

The examples in this chapter are intended for the first time user. They explain in step by step detail the procedures for creating, editing, and saving a Basic program. If a more detailed description of a Basic instruction is required, the reader is referred to the later chapters of this manual which cover the instructions on an individual and in depth basis.

Additional programming examples for the more advanced user are included throughout the manual.

7.1 Getting Started

After the computer has been turned on, the floppy diskette containing Basic and the Cromemco Disk Operating System (CDOS) must be inserted into disk drive A. This is the leftmost disk drive. Start with the diskette supplied by Cromemco. The diskette should be inserted so that the label faces left and the edge with the elongated hole which exposes the surface of the diskette is toward the rear of the machine (disk drive). If there is a door on the disk drive it must be closed after the diskette is inserted.

Next the RETURN key on the console must be depressed several times so that the computer can determine the speed at which the console is sending characters (baud rate). The computer will respond to this by displaying a character on the console (called a prompt) which indicates that it is waiting for the user to type something. If the prompt is a semicolon (;), the user must type B (for Boot) followed by a carriage RETURN to load CDOS into memory. If the prompt is the letter A followed by a period (A.) then the Cromemco Disk Operating System (CDOS) has already been loaded into memory.

7. Programming Examples

In response to the CDOS prompt, type SBASIC followed by a carriage RETURN. After a few seconds the Basic sign on message will be displayed followed by the Basic prompt (>>). Basic has now been loaded into the computer's memory and is waiting for instructions.

One Very Important Point follows:

ALL LINES ENTERED IN BASIC MUST BE TERMINATED BY DEPRESSING THE RETURN KEY.

Basic will never respond to an instruction, accept any input, or complete a command unless it is terminated by a RETURN. If Basic does not seem to be responding as you think it should, make sure that you have properly terminated the current line (with a carriage RETURN.)

Note: It is important to remove disks from the disk drive before turning the power on or off. This will eliminate the possibility of stray bits being written on the disk as the system is powered up or down.

7.1.1 The Command or Immediate Mode

Whenever the Basic prompt (>>) is displayed, Basic is in the Immediate Mode. This means that an instruction (command) can be entered and Basic will respond immediately after the carriage RETURN key is depressed.

The first instruction we will discuss is PRINT. PRINT causes Basic to display the information following the word PRINT. Using only the PRINT instruction, and the standard Arithmetic Operators (+ for addition, - for subtraction, / for division, and * for multiplication), Basic may be used as a calculator.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
7. Programming Examples

<u>Instruction</u>	<u>Explanation</u>
Print 5	This will PRINT the number 5. Since there is nothing following the 5 Basic will add a carriage RETURN and LINE FEED so that the next item which is PRINTed will start in column 0 of the following line.
Print 7 + 4	This will PRINT the result of adding 7 and 4, or 11.
Print "FRED"	This will PRINT the word FRED. Notice that if a group of letters (called a string) are to be PRINTed, they must be enclosed in quotation marks. When a string is enclosed in quotation marks, it is called a string literal. Numbers (called constants) which are to be used in a computation may not be enclosed in quotation marks.
Print	This will PRINT a blank line.
Print 1,2,3,4	This will PRINT the numbers 1 through 4 in four columns across the screen. When commas (,) are used to separate items in a PRINT list, the items are aligned in four columns across the console screen. If semicolons (;) are used, the items are displayed with no intervening spaces.
Print "ANS. = ";75	This will display ANS. = 75 on the console. Notice that a blank was included after the equal sign in the string literal. This blank is considered part of the string literal, just as any other character.

Next we will discuss variables and the Assignment Operator (Instruction). A variable is the name of

7. Programming Examples

a location in the computer's memory. While the name of a variable stays the same, the contents of the variable (location) may vary.

A variable name may be thought of as a label which has been affixed to a box. If we are speaking of an arithmetic variable, then the box would contain a number. The number inside the box is the value of the variable. The value of the variable can be changed while the name of the variable stays the same. Similarly, a string variable would contain a string of characters which could include letters, spaces, numbers, and any other printable characters. This combination of characters would be the value of the string variable.

Arithmetic variable names are composed of a letter (A-Z) followed by any combination of letters, numbers, and apostrophies. Some examples of arithmetic variables are:

Name
Time'of'day
Table'of'values
Social'security'number

String variables may use any name which is a legal arithmetic variable name, however, a dollar sign must immediately follow the name. Some examples of string variables are:

Name\$
Address\$
Vegetable\$
License'plate\$

The assignment operator assigns a value to a variable. The equal sign (=) is used for this purpose. In Basic, the equal sign can be read as, "is assigned the value of." Notice that it does not necessarily denote equality.

7. Programming Examples

<u>Instruction</u>	<u>Explanation</u>
Name\$ = "FRED"	The value of the string literal "FRED" is assigned to the string variable named Name\$. Notice that, although the string literal "FRED" must be enclosed in quotation marks, the quotation marks are not part of the string variable. The quotation marks indicate to Basic that the enclosed characters are to be considered a string literal.
Print Name\$	This instruction will display the value of Name\$ on the console. If this instruction follows the preceding one, FRED will be PRINTed on the console terminal.
P4 = 775	The variable named P4 is assigned the value of 775.
P4 = P4 + 1	P4 is assigned the value of P4 + 1. If this instruction follows the previous one, P4 will have the value of 776.
Text\$ = " IS NO. "	The string variable Text\$ is assigned the value of the string literal " IS NO. ".
Replace\$ = Text\$	The string variable Replace\$ is assigned the value of the string variable Text\$.
Print Name\$; Replace\$; P4	This will PRINT FRED IS NO. 776 (assuming it follows the above assignment instructions).

The next instruction we are going to discuss is the INPUT instruction. When executed, this instruction displays a question mark on the console and waits for the user to supply a number or a string.

7. Programming Examples

<u>Instruction</u>	<u>Explanation</u>
Input Age	This command will display a question mark and wait for the user to enter a number. The number which is entered will be assigned to the variable named Age.
Input Month\$	This command will display a question mark and wait for the user to enter a string (any group of characters, in this case up to a maximum of eleven). The value of the string which is INPUT will be assigned to the string variable named Month\$.

7.1.2 The RUN or Program Execution Mode

Up to this point, all of our examples have been executed as commands or in what is referred to as the immediate mode. Each command was executed as soon as the RETURN key was depressed. Once executed, the entire instruction had to be typed again in order to be re-executed.

Now we shall use these same instructions to write a program. A program consists of statements. A statement is nothing more than an instruction with a line number preceding it. Line numbers are also called statement numbers; the two terms are interchangeable. The line number indicates to Basic that the instruction is to be stored in memory for later execution. A line number (or line name) is also a useful way to refer to a Basic statement if it needs to be changed.

Let's take a moment to cover the rules for naming variables. A variable name must start with a letter (A through Z) which may be followed by up to 30 more characters. These subsequent characters may be letters (A through Z), numbers (0 through 9), or the apostrophe ('). The apostrophe has no different significance than the letters and numbers. It is very useful for breaking up long variable names so that they may be read more clearly. Consider these examples:

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
7. Programming Examples

Timeofday	or	Time'of'day
Firstrecordprimarykey	or	First'record'primary'key
Readarecord	or	Read'a'record
Getabandcblocks	or	Get'a'b'and'c'blocks

Try to make your variable names reflect the meaning of the contents of the variables. This will make your program easier to debug and will also make it easier for another person to understand.

Now we can enter a program:

```
>>100 Input Name'of'person$
>>200 Print Name'of'person$;" is smart!"
```

This program is now current in the User Area. The User Area is the Basic workspace in which a program can be written, EDITed, and RUN. The LIST command displays the contents of the User Area.

The RUN command causes Basic to execute the program in the User Area. Program execution begins with the statement with the lowest line number and continues sequentially.

We can now execute the program:

```
>>Run
? Ed          Basic displays the ? (prompt)
               and the user enters a string.
               Don't forget the carriage
               RETURN at the end of the
               string!

Ed is smart!   Basic PRINTs the string
               variable (Ed) and the string
               literal ( is smart!).

***End***     Basic tells you that it's done
               with the program,

>>           and that it is ready for
               additional instructions.
```

To get a LISTing of your program, type the command LIST in response to the Basic prompt.

7. Programming Examples

```
>>List
```

```
100 Input Name'of'person$  
200 Print Name'of'person$;" is smart!"
```

```
>>
```

7.1.3 Activating the System Printer

The printer (if one is attached to your system) is activated by typing control-P. Control-P is typed by holding down the CTRL key on your terminal (as you would hold the shift key on a typewriter) and simultaneously depressing the P key.

Anytime CTRL-P is typed, the system printer automatically echoes any input, output, or LISTing which is sent to the console.

To LIST the program which is current in the User Area, enter:

```
>>List <CTRL-P> <RETURN>
```

on the console. The LISTing will be sent to the console as well as the printer. When the LISTing has finished, type CTRL-P before anything else is entered. This will ensure that nothing is output to the printer following the LISTing.

Anytime the printer is on and is echoing the console, typing CTRL-P will turn off the printer.

If CTRL-P is inadvertently typed, and there is no system printer, or the printer is off, type a second CTRL-P to resume normal operation. If this is not done, the system will wait for a printer to be ready before displaying any additional output on the console.

7. Programming Examples

7.1.4 Program Editing

Continuing with the same program, suppose we wish to alter a statement.

To change a line it can be typed over again. When Basic recognizes that it already has a line with the same line number, it will replace the old line with the new one. To delete a line, type the line number followed by a carriage RETURN.

We can also use the Basic In Line Editor. The Editor facilitates both minor corrections to long lines and multiple replacements. Suppose we wanted to change line 200:

```
>>Edit 200
```

```
- 200   Print Name'of'person$;" is smart!"
:                                     i very
- 200   Print Name'of'person$;" is very smart!"
:                                     i!!
- 200   Print Name'of'person$;" is very smart!!!"
:<RETURN>
```

```
>>
```

The EDIT command can be given with or without a line number. When used without a line number, each line of the program will be displayed in turn for EDITing. When a line number is given, only the specified statement line will be displayed.

The Editor precedes a line to be EDITed with a dash (-) and prompts the user with a colon (:). Following the colon the user can position the cursor under the part of the line to be EDITed by typing spaces.

In our example above, line 200 was displayed followed on the next line by the colon prompt. Spaces were typed until the cursor was just under the space before smart. Then i (for insert) was typed, followed by the text to be inserted. Notice that an insertion precedes the character above the i. The text to be inserted was followed by a carriage RETURN. Basic displayed the line again, incorporating the change we just made. Next, some exclamation points were added to the end of the line in a similar manner. This time after Basic displayed the corrected line and displayed the

7. Programming Examples

Editor prompt (:) a carriage RETURN was entered to indicate that no further changes were to be made.

While using the EDIT command, the character d may be positioned under any characters in a line which are to be deleted. As many d's must be typed as there are characters to be deleted. After a carriage RETURN is typed, the line will be displayed as EDITed. A carriage RETURN in response to the EDIT prompt will return the user to the Basic monitor.

```
>>List           After making the change, we can
                  list the program to make sure
                  that we changed it properly.
```

```
100 Input Name'of'person$
200 Print Name'of'person$;" is very smart!!!"
```

```
>>              Again, Basic waits for further
                  instructions.
```

Suppose that, instead of executing these statements only once, we wanted them executed several times. We could type RUN each time the program was to be executed, or we could add another statement which would direct Basic back to the beginning of the program each time it finished PRINTing:

```
>>300 Goto 100    This GOTO tells Basic to GOTO
                  statement number 100 and
                  continue execution from there.
```

```
>>List
100 Input Name'of'person$
200 Print Name'of'person$;" is very smart!!!"
300 Goto 100
```

```
>>Run
? Alice          Don't forget the carriage
                  RETURN!
```

```
Alice is very smart!!!
```

```
? Eileen        After it was done with
                  statement 200, statement 300
                  told Basic to go back to
                  statement 100, which gave us
                  another question mark.
```


7. Programming Examples

Eileen is very smart!!!

? This could (and would) go on
 forever. This can be avoided
 by depressing the ESCape key
 (appropriately named). On a
 Cromemco terminal this key is
 located on the upper left of
 the keyboard and is marked
 ESC.

100 ESCAPE

>> Basic tells you what happened,
 and waits for further
 instructions.

7. Programming Examples

7.2 Example Program

Let's write a program to compute a person's age in the year 1985. For this program, we will introduce two new instructions. The first of these is SCR which is short for SCRatch. This instruction clears memory of a program or any statements which may be in the User Area:

```
>>Scr
```

```
>>List           Nothing will be listed, because
                  everything has been SCRatched!
```

```
>>
```

The other is AUTOL which is short for AUTOMATIC Line numbering. This is a convenient feature of Cromemco Structured Basic which allows the programmer to concentrate on the program and forget about entering line numbers. AUTOL is followed by two numbers (called arguments). The first of these indicates the first line number, while the second indicates the increment between line numbers. In the following example, the line numbers are automatically typed by Basic while the program is entered by the user:

```
>>Autol 1000,10
```

```
>>1000 Print"Enter the year of your birth: ";
>>1010 Input Birth'year
>>1020 Print"In 1985 you will be ";1985-Birth'year;
>>1030 Print " years old"
>>1040 Print
>>1050 Goto 1000
>>1060 <RETURN>
>>
```

There are several things about this program which bear discussion. The first is that the AUTOMATIC Line numbering mode may be terminated by entering a carriage RETURN in response to a line number. Line 1020 and 1030 PRINT two strings with a number in the middle. The number is the result of a computation, the numeric variable Birth'year is subtracted from the numeric constant 1985. Line 1040 is included so that there will be a blank line PRINTed between examples.

7. Programming Examples

7.2.1 Listing to a Disk File

The program we have written will stay in memory (in the User Area) as long as we don't execute the SCRatch or BYE command or turn the power off.

We may write the program to a disk file so that it may be retrieved at a later time. Once it has been LISTed on the disk, we may SCRatch the User Area or turn the power off and our program will still be safe on the disk.

The only way a disk file can be destroyed is by ERASing it or by writing another file with the same name to the disk. In the latter case, the second file would be written over the first so that the first file would be destroyed.

The LIST command, as we have been using it, LISTs the contents of the User Area to the console. If the LIST command is followed by a string literal, Basic will interpret the string literal as a file name, and LIST the contents of the User Area to that file. Assuming that our program is still in the User Area, the following command will write the program to a file called FIRST.

```
>>List "First"
```

```
>>
```

The disk should come to life, click once or twice, and another prompt should appear on the console. If Basic has done your bidding, the DIRectory command will reveal a file on the disk called FIRST. To get a list of all the files on the disk enter the command:

```
>> Dir
```

Basic will respond with a list of file names, including FIRST. Now you can shut off the computer without losing your program.

Remember to remove the disk before turning the power off. It is also a good idea to insert the disk only after the power has been turned on. Do not turn the power on or off while the disk is inserted in the machine.

7. Programming Examples

7.2.2 Entering from a Disk File

A program which has been LISTed to a disk file may be re-entered by using the ENTER command.

Before ENTERing any program, whether from a disk file, or from the console, it is a good idea to SCRatch the User Area. Assuming that we are once again (or still) in Basic, we can give the following two commands to clear the User Area and ENTER our program which was LISTed to the disk file FIRST.

```
>> Scr
```

```
>> Enter "First"
```

```
>>
```

Our program may now be LISTed, RUN, or EDITed as we desire.

7. Programming Examples

7.2.3 Using the SAVE and LOAD commands

The SAVE command will write the program which is current in the User Area to a disk file. The program will be in internal machine format unlike a program which is written to disk using the LIST command. The LOAD command is to the SAVE command what the ENTER command is to the LIST command. LOAD allows the user to read a SAVED file into the User Area. Notice that LOAD cannot be used with a LISTed file and that ENTER cannot be used with a SAVED file.

The formats of these commands are:

```
>>Save "Second"  
>>Load "Second"
```

When used with larger programs, SAVE and LOAD can be significantly faster than LIST and ENTER. Also, when the RUN instruction is given with the name of a SAVED file, the specified file will be LOADED and RUN with only that one instruction. Programs may be chained in this fashion, one calling the next, calling the next, etc.

```
>>Run "Second"
```

or

```
9999 Run "Second"
```

The execution of either of these instructions will cause the SAVED program SECOND to be LOADED into the User Area and execution to begin.

7. Programming Examples

7.3 Statistical Analysis Program

```

100 Count=0 : Sum=0 : Sum'of'squares=0
110 Input "Number (9999 to stop): ", Number
120 While Number#9999
130 Count = Count + 1
140 Sum = Sum + Number
150 Sum'of'squares = Sum'of'squares + Number**2
160 Input "Number (9999 to stop): ", Number
170 Endwhile
180 Rem *****
190 Rem All data in, compute statistics.
200 Mean = Sum/Count
210 Temp = Count*Sum'of'squares
220 Variance = ((Temp-(Sum*Sum))/Count)/(Count-1)
230 Standard'deviation = Sqr(Variance)
240 Rem *****
250 Rem All statistics computed, PRINT results
260 Rem Call a user defined function to round
270 Rem results to 4 decimal places.
280 Print "Number", Count
290 Print "Sum", Fnround(Sum)
300 Print "Sum of squares", Fnround(Sum'of'squares)
310 Print "Mean", Fnround(Mean)
320 Print "Variance", Fnround(Variance)
330 Print "Standard'deviation", Fnround(Standard'deviation)
340 Rem
350 Rem
360 Rem
370 Def Fnround(X) = Int(X * 10000. + 0.5)/10000.0
380 End

```

A common use for a computer is the calculation of statistics for a set of data. This program will compute six common statistics for any set of data which the user may enter.

Line 100 initializes three of the program variables. These three variables (Count, Sum, and Sum'of'squares) are used to hold sums of values as the program is executed. They must be set equal to zero at the beginning of the program in order to assure that the values they sum are accurate.

Lines 110 through 170 will continue to accept numbers from the user, keeping track of how many numbers have been INPUT (Count), the sum of the numbers (Sum), and the sum of the squares of the numbers (Sum'of'squares), until the number 9999 is entered. For the purpose of this example, it is

7. Programming Examples

assumed that 9999 is outside the range of numbers which will be INPUT into this program.

Control will remain within the WHILE loop WHILE the variable Number is not equal to (#) 9999. This means that statements 130 through 170 will be executed over and over again WHILE Number is not equal to 9999. As soon as the user enters 9999, the variable Number is assigned the value 9999, the condition for the WHILE loop is false (Number now equals 9999), and control is passed to the statement following the ENDWHILE.

Now the rest of the statistics are computed and all results are PRINTed.

Notice that the user defined function Fnround is used to round the answers to 4 decimal places for PRINTing.

INSTRUCTIONS AND FUNCTIONS

The following chapters describe the instruction set of Cromemco's 32K Structured Basic. Although several of the instructions perform a variety of tasks, an attempt has been made to group the instructions and functions into logical divisions.

The chapter on Program Development encompasses the Basic In Line Editor as well as those instructions which are most frequently used by a programmer while developing or modifying programs. Documentation covers the REMark instruction and Assignment covers the use of the Assignment Operator in the LET and MAT instructions.

The Initialization chapter describes those instructions which set and change the type of numeric representation which is used in a program as well as the DIMension instruction.

Control Structures are the instructions which control the logical flow of a program. This includes conditional and unconditional transfer of control, loops, and program termination.

The Input/Output chapters begin with instructions which may be used to transfer information to and from the console terminal as well as those instructions which are used to read DATA statements from the program itself. A discussion of the theory and use of Data Files precedes the definition of the various Basic instructions which allow the user to control and access these files.

The next chapter covers Arithmetic, Trigonometric, Programmer Defined, and String Functions.

The System and File Status chapter shows the user how to determine and change various system statuses and parameters.

Then Machine Level Instructions which allow the user to interact with Assembly Language subroutines

8. Program Development

are discussed.

The chapter on the Scope of Variables covers the implementation of LOCAL Variables as well as the use of the Common Storage Area for variables.

The section on Procedures discusses the definition of, calling, and returning from Procedures in addition to the parameter passing features of Structured Basic's Procedures. The chapter on Basic-KSAM discusses in depth the need for and theory of Basic-KSAM as well as describing each of the Basic-KSAM instructions.

The final chapters include a Glossary of words which may be unfamiliar to the novice, and an Appendix which describes various aspects of the language which may be of interest to the more advanced user.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
8. Program Development

command: Automatic Line Numbering

format: AUTOL L1,L2

where:

L1 is the starting line number.

L2 is the line number increment.

The AUTOL command provides automatic statement line numbering so that the user does not have to enter a line number for each line when entering a program.

Note:

1. The automatic generation of line numbers may be terminated by pressing the ESCape or RETURN key when the user is prompted for the next line.

Example:

```
>>Autol 100,10

>>100 Rem All line numbers in this
>>110 Rem example are generated by Basic.
>>120 Index,=5
          $
Error 1 -- Syntax
>>120 Index=5
>>130 Rem Notice that after a Syntax Error
>>140 Rem Autol will re-prompt so that
>>150 Rem the error can be corrected.
>>160 Print Index
>>170 End
>>180 <RETURN>
>>
```

8. Program Development

instruction: Bye

format: [Ln] BYE

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The BYE instruction is used to exit from Basic and return to CDOS.

Notes:

1. After the BYE instruction is executed the computer will respond with the current disk drive (the CDOS prompt).
2. BYE will close all files which are OPEN at the time the instruction is executed.

Example:

>>Bye

B.

In this example, the user has typed the BYE command. The next prompt displayed (B.) indicates that the user is in CDOS and that disk drive B is the current drive.

8. Program Development

instruction: Delete Statement Lines

format: [Ln] DELETE L1

[Ln] DELETE L1,

[Ln] DELETE L1,L2

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

L1 is a line number or line name.

If L1 is the only argument and there is no comma following it, then L1 is the only line DELETED.

If L1 is the only argument and the comma is included, L1 through the last line in the program are DELETED.

L2 is an optional line number or line name which indicates the last line to be DELETED. If included, it must be preceded by L1 and a comma.

The DELETE instruction is used to remove statement lines from the program currently in the User Area.

Notes:

1. The DELETE instruction must be the last (or only) instruction on a line.
2. The DELETE instruction must have at least one argument (L1).

8. Program Development

Example:

>>List

```
10  Input Num1,Num2,Num3
20  Sum = Num1+Num2+Num3
30  Print Num1
40  Print Num2
50  Print Num3
60  Print Sum
70  End
```

>>Delete 30,50

>>List

```
10  Input Num1,Num2,Num3
20  Sum = Num1+Num2+Num3
60  Print Sum
70  End
```

Here the DELETE command removed lines 30 through 50 from the program.

8. Program Development

command: Directory

format: DIR

DIR svar

where:

svar is a string variable or a string
literal file reference.

The DIR command corresponds to the CDOS Directory command (see the CDOS User's Manual for a full description). The DIR command lists disk files giving the size of each file in Kilobytes. If the optional file reference is used, it must be enclosed in quotation marks (string literal) or else must be a valid string variable.

Examples:

Dir will list all files on the
current disk.

Dir "A:*.*)" will list all files on drive A.

Dir "*.SAV" will list all files on the
current disk with the file name
extension SAV.

CROMEMCO IN LINE BASIC EDITOR

commands: Change
Edit
Find

format: command

command L1

command L1,

command L1,L2

where:

L1 is an optional line number or name.

If L1 is omitted, all lines of the program are processed.

If L1 is the only argument and there is no comma following it, then L1 is the only line processed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are processed by the command.

L2 is an optional line number or name which indicates the last line to be processed. If included, it must be preceded by L1 and a comma.

Notes:

1. Refer to the chapter on Programming Examples for an example of the use of the Editor.
2. When using terminals in upper case mode, Basic should be set so that it lists in upper case by entering the command:

SET 6,1

8. Program Development

EDIT

EDIT is one of the three commands which are collectively called the Cromemco In Line Basic Editor. The other two are CHANGE and FIND. EDIT lists lines of code one at a time, as specified by the arguments L1 and L2. Each line to be EDITed is preceded by a dash (-) and followed on the next line by a colon (:) prompt. In response to the prompt, the user may type:

1. a carriage RETURN, which leaves the line unchanged and displays the next line for EDITing, or
2. a series of spaces, which will position the cursor under the character which is to be EDITed.

At this point, the user can give the following commands:

- a. D (Delete) - deletes the character above the cursor. Several deletions can be made on one line.
- b. I (Insert) - inserts the string which follows. An insertion can follow one or more deletions. An insertion precedes the character above the I.
- c. K (Kill) - deletes the rest of the line from the current position of the cursor.

A RETURN causes the command line to be executed. The edited line is then typed out and a prompt is given for further changes.

When the line has been changed to the user's satisfaction, a RETURN should be typed in response to the colon(:) prompt.

8. Program Development

FIND

FIND is another of the interactive Editor commands. FIND will locate all occurrences of a string within the set of lines specified by L1 and L2. After the FIND command is given, the Editor will prompt with:

FIND:

In response, the user should enter the string to be located, followed by a carriage RETURN. If a RETURN is entered immediately following the prompt, the Editor will return control to the Basic monitor.

FIND will cause each line containing the string to be printed out with a pointer below the line indicating the occurrence of the specified string.

8. Program Development

CHANGE

The CHANGE command will replace all occurrences of a string within a set of lines specified by L1 and L2. After the CHANGE command is given, the Editor will prompt with:

FROM:

In response, the user should enter the string to be replaced followed by a carriage RETURN. The Editor will then prompt with:

TO:

The user should respond with the replacement string. The Editor will then print the first line containing the string with a pointer below the line indicating the location of the string. The user can then input:

1. a RETURN to reject the CHANGE at that location,
2. the letter (C) to accept the CHANGE, or
3. an asterisk (*) to accept all CHANGES from that point on.

instruction: Enter File

format: [Ln] ENTER svar

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file reference.

The ENTER instruction is used to ENTER a Basic program (in ASCII format) from a disk file or other external device into the User Area.

Notes:

1. ENTER will read a program which was written to the disk by the LIST instruction. The program will be read into the User Area. ENTER will not read a program which was written to the disk by the SAVE instruction.
2. ENTER does not delete statement lines from the program which is currently in the User Area. ENTER does replace lines in the current program with lines from the file being ENTERed if the line numbers are the same.
3. Efficient use of memory following an overlay results if the ENTERed program, to as great an extent as is possible, replaces lines in the current program and does not add new line numbers.
4. The ASCII ESCape character (1BH) or CTRL-Z (1AH) are used as the end of file mark by Basic. ENTER looks for either of these characters to determine the end of the program.

8. Program Development

instruction: List Current Program

format: [Ln] LIST

[Ln] LIST L1

[Ln] LIST L1,

[Ln] LIST L1,L2

[Ln] LIST svar

[Ln] LIST svar,L1

[Ln] LIST svar,L1,

[Ln] LIST svar,L1,L2

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

L1 is an optional line number or line name.

If L1 is omitted, all lines of the program are LISTed.

If L1 is the only argument and there is no comma following it, then L1 is the only line LISTed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are LISTed.

L2 is an optional line number or line name which indicates the last line to be LISTed. If included, it must be preceded by L1 and a comma.

8. Program Development

 svar is a optional string variable or
 a string literal file reference
 denoting the destination of the
 LISTing. If omitted, the
 LISTing is displayed on the
 console.

The LIST instruction is used to LIST one or more statement lines from the User Area to the console, a disk file, or another file device in ASCII format. The instruction may be used to output an entire program, a block of statement lines within a program, or a single statement line. The formats of LIST which do not use svar will direct the output to the console.

Notes:

1. A program which has been LISTed to a disk can be read back into the User Area using the ENTER instruction. It can not be read back using LOAD or RUN.
2. LISTed files are compatible between different versions of Cromemco Structured Basic. SAVED files are not necessarily compatible in this manner.
3. The ASCII ESCape character (1BH) is used as the end of file mark by Basic. LIST outputs this character at the end of a program.

8. Program Development

instruction: List Variables

format: [Ln] LVAR

[Ln] LVAR file-ref

where: Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref is an optional string variable or string literal file-reference denoting the destination of the list of variables. If omitted, the listing is displayed on the console.

The LVAR instruction lists all variables, functions, procedures, and alphanumeric line names. The abbreviations used in the list are:

INT	integer variable
SFP	short floating point variable
LFP	long floating point variable
LBL	alphanumeric label, line name
FUN	function name
PROC	procedure name
\$	string variable
(*)	list or matrix

The current value of each scalar arithmetic variable is also displayed.

Notes:

1. If a variable, function, PROCEDURE name or line name is used in a Basic program, and then the line containing that item is deleted, the item will still appear in the list produced by LVAR. The LIST can be updated by Listing the program to a file, SCRatching the User Area, ENTERing the program, and then using LVAR.
2. If a line name, PROCEDURE name or function has been defined in the program, its line number will be printed after its type. If a line name, PROCEDURE name or function has been used

8. Program Development

(e.g., GOTO Start'over) but not yet defined,
no entry will follow the type.

8. Program Development

instruction: Load Program

format: [Ln] LOAD svar

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file reference.

The LOAD instruction is used to LOAD a Basic program (in internal machine format) from a disk file into the User Area.

Notes:

1. LOAD will read a program which was written to the disk by the SAVE instruction. The program will be read into the User Area. The program must have been SAVED under the version of Basic and the version and size of the Cromemco Disk Operating System under which Basic is currently being run.

Files which are written to the disk by the LIST instruction are compatible between different versions of Cromemco Structured Basic, files written using the SAVE instruction are not.

2. The LOAD instruction resets (clears) all variables, string variables, and matrices.
3. LOAD resets the trigonometric mode to RADians.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
8. Program Development

command: Renumber Statement Lines

format: RENUMBER

RENUMBER L1

RENUMBER L1,L2

RENUMBER L1,L2,L3

RENUMBER L1,L2,L3,

RENUMBER L1,L2,L3,L4

where:

L1 is an optional starting line number in the RENUMBERed program.

L2 is an optional line number increment in the RENUMBERed program.

L3 is an optional line number or line name in the original program.

If L3 is omitted, all lines of the program are RENUMBERed.

If L3 is not followed by a comma then L3 is the only line which is RENUMBERed.

If L3 is followed by a comma, L3 through the last line in the program are RENUMBERed.

L4 is an optional line number or line name which indicates the last line in the original program to be RENUMBERed.

The RENUMBER command alters the statement numbers in the current program.

Notes:

1. The default value for the RENUMBER command is a starting line number (L1) of 10 and an

8. Program Development

increment value (L2) of 10.

2. If only the first parameter (L1) is specified, the second parameter (L2) assumes the same value as the first. In the example below, the command `RENUMBER 100` is equivalent to the command `RENUMBER 100,100`.
3. The `RENUMBER` instruction alters line numbers imbedded in the entire program in `GOTO`, `GOSUB`, and `IF-THEN` statements to conform to the `RENUMBERED` statements. This will affect a line which is not `RENUMBERED` if the line contains a reference to a `RENUMBERED` line.
4. `RENUMBER` cannot normally be used to re-order or rearrange sections of a program relative to other sections. If line numbers are `LISTED` out of order after the `RENUMBER` instruction is given, follow the procedure below (5a-d) to rearrange the lines in numeric order.
5. The `RENUMBER` command will include `DELETED` statement numbers in the sequence of `RENUMBERED` statements. If this presents a problem (such as one or more statement numbers being omitted) the following procedure will correct the problem:
 - a) `LIST` the program (do not `SAVE` it) to a temporary disk file.
 - b) `SCRATCH` the User Area.
 - c) `ENTER` the temporary disk file.
 - d) `RENUMBER` as desired.

8. Program Development

Examples:

>>List

```

11   Input Alpha
24   Input Beta
37   Print Alpha*Beta
50   Goto 11
63   End

```

>>Renumber (default parameters are 10,10)

>>List

```

10   Input Alpha
20   Input Beta
30   Print Alpha*Beta
40   Goto 10
50   End

```

>>Renumber 100 (if L2 is not specified, L1=L1)

>>List

```

100   Input Alpha
200   Input Beta
300   Print Alpha*Beta
400   Goto 100
500   End

```

>>Renumber 100,10

>>List

```

100   Input Alpha
110   Input Beta
120   Print Alpha*Beta
130   Goto 100
140   End

```

```

>>Renumber 1000,150,120,
                                (Renumber all lines
                                from line 120 to
                                the end of the
                                program.)

```

8. Program Development

>>List

```
100  Input Alpha
110  Input Beta
1000 Print Alpha*Beta
1150 Goto 100
1300 End
```

>>Renumber 1000,1,110,1150

(Renumber lines 110 through 1150 in the current program. The new line numbers will start at 1000 and use an increment of 1.)

>>List

```
100  Input Alpha
1000 Input Beta
1001 Print Alpha*Beta
1002 Goto 100
1300 End
```

8. Program Development

instruction: Run Program

format: [Ln] RUN

[Ln] RUN svar

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file reference.

The RUN instruction directs the computer to execute a program starting at the lowest numbered line.

If svar is omitted, the program which is current in Partition zero of the User Area is executed.

If svar is included, it must be the name of a SAVED program. This program will be LOADED into Partition zero of the User Area and executed.

Notes:

1. The RUN instruction, if given with a file reference, must reference a program which has been SAVED under the version of Basic which is currently being used.

LISTed files are compatible between different versions of Cromemco Structured Basic, SAVED files are not.

2. The RUN instruction resets or clears all variables, string variables, and matrices. It also sets the trigonometric mode to RADians.
3. If Partition zero is not the Current Partition (after a USE instruction or if execution was terminated in another Partition), RUN will cause Partition zero to become the Current Partition and execution to begin in Partition zero.
4. RUN resets ON ERROR and ON ESCAPE instructions to their default modes. This means that run-

8. Program Development

time non-fatal errors as well as the use of the ESCape key will cause a running program to abort and Basic to display a message.

5. RUN sets the variable mode to that which was last specified. The default mode is Long Floating Point.

Refer to the Appendix, Areas of User Interest, if it is necessary to change the default variable mode.

8. Program Development

instruction: Save Program

format: [Ln] SAVE svar

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file reference.

The SAVE instruction is used to SAVE the current program on a disk or other file device in internal machine format.

Note:

1. A program which has been written to the disk using the SAVE instruction can be read back using the LOAD or RUN instructions. A SAVED program can only be LOADED or RUN with the same version of Basic it was SAVED under.

8. Program Development

instruction: Scratch User Area

format: [Ln] SCR

where:

Ln is an optional line number. If
Ln is included, the instruction
is executed at run time.
Otherwise it is executed
immediately.

The SCRatch instruction deletes the current program
from the User Area.

Notes:

1. The programmer should keep in mind that the
 SCRatch command erases everything in all
 Partitions of the User Area and that SCRatched
 programs cannot be recovered.
2. Once the work space has been SCRatched, the
 user may input a new program or access a SAVED
 or LISTed program which has been stored on the
 disk.
3. SCRatch sets the trigonometric mode to
 RADians.
4. SCRatch resets the variable mode to the
 default mode, normally the Long Floating Point
 mode.
5. SCRatch does not reset the ECHO or ON ESCape
 mode.
6. SCRatch CLOSEs all OPEN files before
 performing the actual SCRatch. If any file
 cannot be CLOSEd (because of disk or I/O
 problems) some other files may be left OPEN
 and the User Area will not be SCRatched. The
 other files may be CLOSEd via the CLOSE\n\
 instruction.

8. Program Development

Example:

>>List

```
10    X=4
20    Input Y
30    Z=X*2+Y
40    Print Z
50    End
```

>>Scr

>>List

>>

In the above example, all statement lines are deleted from memory. The user can now input a new program.

8. Program Development

instruction: Enable Trace Option

format: [Ln] TRACE

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The TRACE instruction sets the TRACE mode so that the user can follow the execution of a program line by line. When in the TRACE mode Basic will list the line number of each statement as it is being executed. Statement line numbers will be enclosed in angle brackets.

Example:

>>List

```
10 Trace
20 Input Number
30 Print"This is ";Number
40 Let Number1=Number+1
50 Print Number1
60 End
```

>>Run

<20>

? 10

<30>

This is 10

<40>

<50>

11

<60>

60 End

8. Program Development

instruction: Disable Trace Option

format: [Ln] NTRACE

where:

Ln is an optional line number. If
Ln is included, the instruction
is executed at run time.
Otherwise it is executed
immediately.

The NTRACE instruction resets the TRACE mode so
that line numbers are not displayed during program
execution.

9. Documentation

instruction: Remark

format: [Ln] REM text

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

text is any string of printable characters.

The REM instruction is used to insert REMarks or comments in a program.

Notes:

1. REM statements included in a Basic program are ignored when the program is executed but are output exactly as entered when the program is LISTed.
2. REM statements occupy space in the User Area. With some long programs, or those with large lists or matrices, it may be necessary to minimize the use of REM statements in order to accommodate the program.
3. Any grammatical or typing mistakes which are made when inputting a REM statement will not generate an error message and will be output precisely as they appear in the statement line.
4. The programmer is encouraged to use REM statements liberally throughout a program to describe program operation. These REMarks can be particularly helpful to any one who wishes to use or modify a program written by another person.
5. Multiple spaces in REMark instructions occupy no more User Area than do single spaces. For this reason, the use of multiple spaces in REMarks is encouraged when it will improve readability.

10. Assignment Instructions

instruction: Let

format: [Ln] LET var = exp

or

[Ln] var = exp

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

var is a numeric or string variable or a reference to an element of a matrix.

exp is the value to be assigned to var. It may be any expression, variable, constant, function, string variable, or string literal.

The LET instruction is used to assign a value to a given numeric variable, string variable, or element of a matrix. The equal sign (=) is called the assignment operator. Refer to section 6.2.

Notes:

1. When a string variable is used with a LET instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to section 5.2.3, Referencing String Variables, for a complete discussion of the subject. The LET instruction will not move more characters than can be accepted by the destination string or substring which is being referenced.
2. In Basic, the equal sign is also used as a relational operator. Refer to section 6.3.
3. A string variable or literal may only be

10. Assignment Instructions

assigned to a string variable while any expression or function yielding a numeric result may only be assigned to a numeric variable.

4. Basic is designed to allow the user to assign values to variables without entering LET each time. This capability is called implied LET.

LET and implied LET instructions execute with equal speed.

Examples:

<u>LET Instructions</u>	<u>Equivalent Implied LET Instructions</u>
Let Arm = 45	Arm = 45
Let Bat = Arm + 10	Bat = Arm + 10
Let Text\$ = "Basic"	Text\$ = "Basic"
Let Total = Part1 + Part2	Total = Part1 + Part2
Let Element = Inversion(10,5)	Element = Inversion(10,5)

10. Assignment Instructions

instruction: Matrix Initialization

format: MAT mvar = aexp

where:

aexp is an arithmetic expression,
variable, or constant.

mvar is a MATrix arithmetic variable.

The MAT instruction is used to set all elements in a MATrix (M) equal to the value of the arithmetic expression (aexp).

Notes:

1. MATrix M must be explicitly dimensioned.

Example:

>>List

```

10 Dim Array(4)
20 Read Array(1),Array(2),Array(3),Array(4)
30 Data 20,21,22,23
40 Print Array(1),Array(2),Array(3),Array(4)
50 Mat Array=0
60 Print Array(1),Array(2),Array(3),Array(4)
70 Mat Array=1
80 Print Array(1),Array(2),Array(3),Array(4)
90 End

```

>>Run

```

20          21          22          23
0           0           0           0
1           1           1           1
***90 End***

```


11. Initialization

instruction: Degree Mode

format: [Ln] DEG

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The DEG instruction sets the trigonometric calculation mode to DEGREE.

Note:

1. RUN, SCRatch, and LOAD will automatically reset the trigonometric calculation mode to RADian.

11. Initialization

instruction: Dimension

format: [Ln] DIM svar(aexp-1)

[Ln] DIM avar(aexp-1)

[Ln] DIM avar(aexp-1,aexp-2)

[Ln] DIM avar(aexp-1,aexp-2,aexp-3)

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

avar is a matrix arithmetic variable.

svar is a string variable.

aexp1-3 are arithmetic expressions, variables or constants.

The DIM instruction is used to define the size of a matrix or a string variable. Cromemco Basic permits the user to define one, two, or three DIMensional matrices.

Notes:

1. A DIMensioned numeric matrix variable can have the same name as any other numeric variable. A DIMensioned string variable must have the name of a string variable.
2. If a matrix or string variable is not explicitly dimensioned in a program, the default value of 10 (11 elements, numbered 0 through 10) will be automatically assigned to a singly subscripted matrix or string variable.

Doubly and triply subscripted matrices will generate an error message if not explicitly dimensioned.

3. The maximum size of any matrix is only restricted by the amount of available memory. Any single DIMension may not exceed 16382.

11. Initialization

4. The DIMension of a string variable may not exceed 32766.
5. The first element in a matrix is numbered 0 (zero indexing).

instruction: Integer Mode

format: [Ln] IMODE

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The IMODE instruction changes the default mode of all variables to the Integer mode.

Notes:

1. This instruction takes effect only after the execution of a RUN instruction after the IMODE instruction has been given. See the following example.
2. Integer variables occupy 2 bytes and must be within the range +32767 to -32768.
3. This instruction will be overridden by the LONG and SHORT instructions.

Example:

1 Imode : X=2.5 : If X=2.5 Then Run

This line, appearing as the first line of a program, will ensure that the interpreter is in the Integer mode. If the line is encountered while the interpreter is not in the Integer mode, the IMODE instruction will be given, X will be set equal to 2.5 (a non integer number), and if X=2.5 (as will be the case if SFMODE or LFMODE are current) the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will be given after the IMODE instruction and the current mode will be integer. When the value 2.5 is assigned to X, X will be an integer variable so that 2.5 will be rounded and X will have the value of 3. Then X=2.5 will be false and control will be transferred to the next line.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
11. Initialization

instruction: Integer Variable

format: [Ln] INTEGER avar, mvar(x),...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

avar is a scalar arithmetic variable

mvar is a matrix arithmetic variable.

x is the optional DIMension of mvar.

The INTEGER instruction is used to set a given variable to the INTEGER mode.

Notes:

1. DIMensioning may be done via the INTEGER instruction.
2. The INTEGER instruction must be executed before the variable is referenced for the first time.
3. INTEGER variables occupy 2 bytes and must be within the range +32767 to -32768.
4. This instruction overrides the SFMODE and LFMODE instructions.

11. Initialization

instruction: Long Floating Point Mode

format: [Ln] LFMODE

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The LFMODE instruction is used to set all variables within a program to the Long Floating Point mode.

Notes:

1. This is the standard default mode for all variables and arithmetic operations.
2. This instruction takes effect only after the execution of a RUN instruction after the LFMODE instruction has been given. See the following example.
3. Long Floating Point variables occupy 8 bytes and must be within the range $+9.99E+62$ to $+9.99E-65$. They have an accuracy of 14 digits.
4. This instruction will be overridden by the INTEGER and SHORT instructions.

Example:

```
1 Lfmode : X=0.12345678 : If X<>0.12345678 Then Run
```

This line, appearing as the first line of a program, will ensure that the interpreter is in the Long Floating Point mode. If the line is encountered while the interpreter is not in the Long Floating Point mode, the LFMODE instruction will be given, X will be set equal to 0.12345678 (a number which cannot be represented in either Short Floating Point or Integer modes), and if X does not equal 0.12345678 (as will be the case if SFMODE or IMODE are current) the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will be given after the LFMODE instruction and the current mode

11. Initialization

will be Long Floating Point. When the value 0.12345678 is assigned to X, X will be a Long Floating Point variable with the value of 0.12345678. Then $X \neq 0.12345678$ will be false and control will be transferred to the next line.

11. Initialization

instruction: Long Variable

format: [Ln] LONG avar, mvar(x),...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

avar is a scalar arithmetic variable.

mvar is a matrix arithmetic variable.

x is the optional DIMension of mvar.

The LONG instruction is used to set a given variable to the Long Floating Point mode.

Notes:

1. DIMensioning may be done via the LONG instruction.
2. The LONG instruction must be executed before the variable is referenced for the first time.
3. Long Floating Point variables occupy 8 bytes, have an accuracy of 14 digits, and must be within the range $\pm 9.99\text{E}+62$ to $\pm 9.99\text{E}-65$.
4. This instruction overrides the IMODE and SFMODE instructions.

11. Initialization

instruction: Radian Mode

format: [Ln] RAD

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The RAD instruction sets the RADian mode for trigonometric calculations.

Note:

1. SCRatch, RUN, and LOAD will automatically reset the trigonometric mode to RADian.

11. Initialization

instruction: Short Floating Point Mode

format: [Ln] SFMODE

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The SFMODE instruction is used to set all variables within a program to the Short Floating Point mode.

Notes:

1. This instruction takes effect only after the execution of a RUN instruction after the SFMODE instruction has been given. See the following example.
2. Short Floating Point variables occupy 4 bytes, have an accuracy of 6 digits, and must be within the range $\pm 9.99E+62$ to $\pm 9.99E-65$.
3. This instruction will be overridden by the LONG and INTEGER instructions.

Example:

1 Sfmode : X=0.90000001 : If X<>0.9 Then Run

This line, appearing as the first line of a program, will ensure that the interpreter is in the Short Floating Point mode. If the line is encountered while the interpreter is not in the Short Floating Point mode, the SFMODE instruction will be given, X will be set equal to 0.90000001 and if X is not equal to 0.9 (as will be the case if IMODE or LFMODE are current) the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will have been given after the SFMODE instruction and the current mode will be Short Floating Point. When the value 0.90000001 is assigned to X, X will be a Short Floating Point variable so that 0.90000001 will be rounded and X will have the value of 0.9. Then X<>0.9 will be false and control will be transferred to the next line.

11. Initialization

instruction: Short Variable

format: [Ln] SHORT avar, mvar(x),...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

avar is a scalar arithmetic variable.

mvar is a matrix arithmetic variable.

x is the optional DIMension of mvar.

The Short instruction is used to set a given variable to the Short Floating Point mode.

Notes:

1. DIMensioning may be done via the SHORT instruction.
2. The SHORT instruction must be executed before the variable is referenced for the first time.
3. Short Floating Point variables occupy 4 bytes, have an accuracy of 6 digits, and must be within the range $\pm 9.99 \times 10^{+62}$ to $\pm 9.99 \times 10^{-65}$.
4. This instruction overrides the IMODE and LFMODE instructions.

12. Control Structures

command: Continue Program Execution

format: CON

The CON command CONTinues program execution after a program is interrupted by a STOP statement, a program error, or the ESCape key.

Notes:

1. Program execution will commence with the line following the statement at which the program stopped.
2. If program execution stopped because of a program error, the error can be corrected and the CON command used to continue execution from the line following the one where the error occurred.
3. If the user wishes to re-execute the line in error, GOTO (as a command) should be used.

12. Control Structures

statement: End Program Execution

format: Ln END

where:

Ln is a line number.

The END statement halts program execution and causes Basic to return to the command mode.

Notes:

1. Unlike the STOP statement, program execution may not be CONTinued after an END statement has been executed.
2. Upon execution of the END statement, Basic displays a message on the console indicating the line number of the END statement which caused the program to halt.

Example:

>>List

```
100 Print "Tomorrow and tomorrow and tomorrow..."
200 End
```

>>Run

```
Tomorrow and tomorrow and tomorrow...
***200 End***
```

>>

12. Control Structures

instructions: For-Next Loop

```

format:  [Ln] FOR avar=aexp1 To aexp2 [Step aexp3]
          .
          .
          [program instructions]
          .
          .
          [Ln] NEXT avar

```

where:

Ln are optional line numbers. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

avar is a non-subscripted numeric variable. This is the index variable.

aexp1-3 are arithmetic expressions, variables, or constants. They may not be string literals or string variables. These are the FOR-NEXT loop parameters.

aexp-1 is the initial value.

aexp-2 is the final value.

aexp-3 is the step value.

The FOR-NEXT instructions are used to repeat a part of a Basic program a specified number of times. During the execution of a FOR-NEXT loop, an index (avar) is maintained. When this index becomes equal to (or greater than, or less than) the final value (aexp-3), control is transferred to the instruction following the NEXT instruction.

Notes:

1. The following sequence defines the execution of a FOR-NEXT loop:
 - A. The expressions aexp-1, aexp-2, and aexp-3 are evaluated. If aexp-3 is omitted it is given a value of +1 (if the step value is not specified it is assumed to be +1).

- B. The index variable (avar) is set equal to the initial value (aexp-1).
- C. The instructions following the FOR and preceding the NEXT are executed.
- D. The step value is added to the index variable (avar = avar + aexp-3).
- E. If the step value (aexp-3) is positive and the index variable (avar) is greater than the final value (aexp-2) then the condition for termination of the FOR-NEXT loop has been met and control is transferred to the instruction following the corresponding NEXT instruction.

If the step value (aexp-3) is negative and the index variable (avar) is less than the final value (aexp-2) then the condition for termination of the FOR-NEXT loop has been met and control is transferred to the instruction following the corresponding NEXT instruction.

Otherwise execution of the loop continues with C above.

- 2. The step (aexp-3) portion of the FOR instruction is optional. If a step value is not specified, Basic assumes a value of +1 for aexp-3.
- 3. Basic programs will execute significantly faster if the loop parameter variables are declared as type Integer.
- 4. FOR-NEXT loops may be nested within a program. It is important to keep in mind, however, that each FOR instruction and its corresponding NEXT instruction must be completely contained within any larger loop.
- 5. Program LISTings have FOR-NEXT loops indented for clarity.

12. Control Structures

Examples:

>>List

```

10  Rem Demonstration Program
20  For Index'var = 0 TO 10 Step 2
30  Sum = Index'var+1
40  Print Sum;
50  Next Index'var
60  End

```

>>Run

1357911***60End***

In the above example, upon entering the FOR-NEXT loop defined by lines 20 through 50, the loop parameters are evaluated and set equal to:

```

0 (aexp-1, the initial value)
10 (aexp-2, the final value)
2 (aexp-3, the step value)

```

The index variable is then set equal to the initial value (Index'var=0). Execution continues with lines 30 and 40 where the Index'var maintains the value assigned to it by the FOR instruction. At line 50 the index variable is tested to see if it is greater than the final value (is Index'var > 10?). In this case 0 is not greater than 10, so execution continues with line 20. The step value is added to the index variable (Index'var = Index'var + aexp-3). This continues until (Index'var > aexp-2). Then control is transferred to statement 60.

Correct Nesting Format:

```

10  For Var1 = 1 TO 50 Step 2
20  For Var2 = 1 TO 30 Step 5
30  For Var3 = 1 TO 10 Step 1
    .
    .
    .
100 Next Var3
110 Next Var2
120 Next Var1

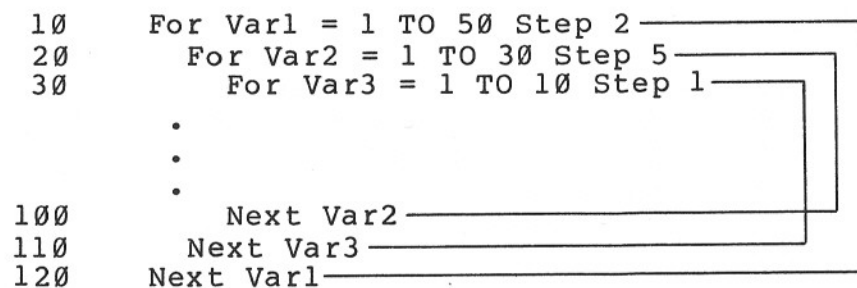
```

Illegal nesting occurs when FOR-NEXT loops overlap. The following example will generate a run time error.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
12. Control Structures

Incorrect Nesting Format

```
10    For Var1 = 1 TO 50 Step 2
20      For Var2 = 1 TO 30 Step 5
30        For Var3 = 1 TO 10 Step 1
          .
          .
          .
100      Next Var2
110      Next Var3
120    Next Var1
```



12. Control Structures

instruction: Gosub-Return

format: [Ln] GOSUB n

·
·
n [program instructions]
·
·

[Ln] RETURN

where:

Ln are optional line numbers. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is the line number or line name of the first statement of the subroutine to which control is transferred.

The GOSUB instruction transfers control to a subroutine.

When, during the execution of the subroutine, a RETURN instruction is executed, control is passed to the instruction following the GOSUB instruction which called the subroutine.

Notes:

1. In Basic, subroutines may be fully enclosed or nested in other subroutines.

When nesting subroutines, remember that the RETURN instruction will take you back to the last GOSUB and start execution of the instruction immediately following. Improperly nested GOSUB-RETURN instructions will generate runtime error messages.

A nested subroutine is executed after the GOSUB statement and before the RETURN statement of the subroutine in which it is enclosed.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
12. Control Structures

Example:

>>List

```
10  Rem Example program, GOSUB
20  Gosub Demo'print
30  Print "Program Over"
40  End
50  *Demo'print : Print "This is a subroutine"
60  Print "which demonstrates the"
70  Print "Gosub statement"
80  Print
90  Return
```

>>Run

This is a subroutine
which demonstrates the
Gosub statement

Program Over
40 End

12. Control Structures

instruction: Gosub-Retry

format: [Ln] GOSUB n

·
·
n [program instructions]

·
·
[Ln] RETRY

where:

Ln are optional line numbers. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is the line number or line name of the first statement of the subroutine to which control is transferred.

The GOSUB instruction transfers control to a subroutine.

The RETRY instruction is similar to the RETURN from a subroutine instruction except that it re-executes the instruction which called the subroutine.

Notes:

1. RETRY is to be used in conjunction with the ON ERROR-GOSUB error trapping instruction only.

When used with an error trap, the condition which caused the error can be fixed and the statement which caused the error will be re-executed.

2. When used in place of RETURN in a subroutine which was called by a standard GOSUB instruction, RETRY will cause Basic to repeatedly execute the subroutine without end.

12. Control Structures

Example:

```
100 Procedure .Open'file (File'name$)
110   Rem
120   Rem This Procedure OPENS a file on channel 1.
130   Rem If the file does not exist it will be CREATED
140   Rem before it is OPENed.
150   Rem
160   On Error Gosub File'create : Rem Set error trap
170   Open\1\File'name$
180   On Error Stop : Rem Reset error trap.
190   Endproc
200   Rem
210   Rem
220   Rem Subroutine FILE'CREATE
230   Rem If called because of an error 134
240   Rem (Cannot Open File) this routine will CREATE
250   Rem the file. If called because of any other
260   Rem error, the Basic error flag will be set,
270   Rem the error trap will be reset, and the
280   Rem Procedure aborted.
290   Rem
300   *File'create
310   Error'number=Sys(3)
320   If Error'number=134 Then Do
330     Create File'name$
340   Else
350     On Error Stop
360     Errproc
370   Enddo
380   Retry
390   End
```


CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
12. Control Structures

instruction: Goto

format: [Ln] GOTO n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is the line number or line name line number of the statement to which control is transferred.

The GOTO instruction unconditionally transfers control to the statement line specified by n.

When used as a statement, GOTO interrupts the normal execution sequence of program statements and transfers control to the specified statement.

When used as a command GOTO will cause execution of a program to start with the specified statement.

Notes:

1. When used with the IF-THEN instruction and a line number, the words GOTO are optional:

If Bool=0 Then Goto 50

or

If Bool=0 Then 50

are equivalent and are both legal instructions.

2. When execution of a program is initiated or continued using a GOTO command, no variable initialization takes place.

Example:

>>List

```
10 Input Number
20 If Number<0 Then 200
30 Root = Sqr(Number)
40 Print "The square root of ";Number;" is ";Root
50 Goto 210
200 Print "This yields an imaginary number"
210 End
```

>>Run

? -2

This yields an imaginary number

210 End

>>Run

? 9

The square root of 9 is 3

210 End

In statement 20 the words GOTO are omitted. This statement causes control to be transferred to statement 200 if the condition ($A < 0$) is true. Statement 50 unconditionally transfers control to statement 210.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
12. Control Structures

instruction: If-Then

format-1: [Ln] IF exp THEN n

format-2: [Ln] IF exp THEN instruction

[Ln] IF exp THEN instruction:instruction:...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

exp is a relational or arithmetic expression, an arithmetic variable, or a constant.

n is the line number or line name of the statement to which control is transferred.

instruction is any Basic instruction except FOR, NEXT, REM or DATA.

The IF-THEN instruction evaluates exp to false (=0) or true (not equal to 0).

If format-1 is used, control is transferred to the specified statement if exp is evaluated as true, and to the next sequential statement if exp is false.

If format-2 is used, and if exp is true, the instructions remaining on the same line are executed before control is transferred to the next sequential statement. If exp is false, control passes to the next sequential statement line.

Notes:

1. In a relational expression, a relational operator (=, <, >=, <=, <>, or >) is used to compare two expressions or values. Refer to sections 6.3 and 6.4 for a discussion of relational and Boolean operators.
2. No instruction may follow format-1 of the IF-THEN instruction on the same line.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
12. Control Structures

3. IF-THEN instructions may be nested using format-2. For example:

```
100 If A = B Then If G$ = "N" Then 500
```

is a legal statement.

4. Also refer to the IF-THEN-ELSE instructions.

Examples:

```
100 *In : Input Part'number
110 If Part'number=0 Then @"must be non-zero" : Goto In
120 Print Part'number
130 End
```

In this example, the computer outputs a prompt (?) to which the user responds with a number. IF the number is non-zero (Part'number=0 is false), control will be passed to line 120, the number will be printed, and execution of the program will terminate. IF the number is zero (Part'number=0 is true), the part of line 110 after THEN will be executed, printing out the message and returning control to the line labeled In (line 100) which will request another number from the user.

```
100 *Again : Input "First Name: ", Name$
200 If Name$ = "FRED" Or Name$ = "Fred" Then Goto Finish
300 Goto Again
400 *Finish : Print "So you are Fred!"
500 End
```

This program will continue to prompt the user with "First Name:" until the user responds with either Fred or FRED. The IF-THEN instruction includes a compound relational expression:

```
Name$ = "FRED" Or Name$ = "Fred"
```

This expression is evaluated as true (=1) if either FRED (all upper case) OR Fred (Upper case F, lower case red) is entered. This type of checking is very useful in interactive programs where a variety of user responses are to be allowed.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
12. Control Structures

statements: If-Then-Else

```
format: Ln IF exp THEN DO
        .
        .
        .
        [Ln ELSE]
        .
        .
        .
        Ln ENDDO
```

where:

Ln are line numbers

exp is a relational or arithmetic
 expression.

If exp evaluates to true (not equal to zero) then the program instructions following DO are executed. If, in addition, ELSE is coded, ELSE will transfer control to the ENDDO statement.

If exp evaluates to false (equal to zero) then control is transferred to ELSE (if ELSE is used) or to ENDDO (if ELSE is not used).

Example:

The following program will request a line of text from the user, and then count the number of words and non-blank characters in the line. The average number of characters per word will be computed and displayed and then the user will be prompted for another line of text. Entering a carriage RETURN in response to the request for INPUT will terminate the program.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
12. Control Structures

```
100 Dim Buffer$(100)
110 Dim Blank$(0) : Blank$=" "
120 Rem initialize blank character flag
130 Rem word counter
140 Rem character counter
150 Last'char'was'blank=0
160 Number'of'words=1
170 Number'of'char=0
180 Rem prompt user
190 Input"Enter line: ",Buffer$
200 Rem check for user termination (null buffer)
210 Length'of'buffer=Len(Buffer$)
220 If Length'of'buffer=0 Then Goto 510
230 Rem
240 Rem
250 Rem loop through buffer
260 For Index=0 To Length'of'buffer-1
270 Rem check for blank character
280 If Buffer$(Index,-1)=Blank$Then Do
290 Rem check if the last character was a blank
300 If Last'char'was'blank=0 Then Do
310 Rem if not, increment word counter and
320 Rem set flag
330 Number'of'words=Number'of'words+1
340 Last'char'was'blank=1
350 Enddo
360 Rem if not a blank, increment character counter
370 Rem and reset flag
380 Else
390 Number'of'char=Number'of'char+1
400 Last'char'was'blank=0
410 Enddo
420 Next Index
430 Rem
440 Rem
450 Rem display results and return for another user entry
460 @"Number of non-blank characters is ";Number'of'char
470 @"Number of words is ";Number'of'words
480 Avg=Number'of'char/Number'of'words
490 @"Average number of characters/word is ";Avg
500 Goto 150
510 End
```

12. Control Structures

instructions: On-Goto
On-Gosub

format: [Ln] ON aexp GOTO n1,n2,...,ni

[Ln] ON aexp GOSUB n1,n2,...,ni

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

aexp is an arithmetic expression, variable, or constant

n1-i are line numbers and/or line names of the statements to which control can be transferred.

The ON-GOTO and ON-GOSUB instructions transfer control to any one of several lines in a program based on the value of the expression (aexp) contained in the instruction.

Refer to the GOTO and GOSUB instructions for more information.

Notes:

1. When aexp is evaluated, if it is equal to one, control will be passed to the statement n1; if it is equal to two, control will be passed to the statement n2; if it is equal to i, control will be passed to the statement ni.
2. If aexp evaluates to a non integer number the value of the number will be rounded to the nearest integer for the purpose of these instructions.
3. If aexp is evaluated as less than one or greater than i the instruction will be ignored and control will pass to the next sequential instruction.
4. If n1, n2, or ni is a nonexistent line, and if control is transferred to that line, a fatal run time error will be generated.

5. No instruction may follow an ON-GOTO or ON-GOSUB instruction on the same line.

12. Control Structures

statements: Repeat-Until Loop

```
format: Ln REPEAT
        .
        .
        .
        Ln UNTIL exp
```

where:

Ln is a line number.

exp is an arithmetic or relational expression.

The REPEAT structure is used to REPEAT a set of Basic program instructions UNTIL exp is evaluated as true (not equal to zero).

Notes:

1. The Basic program instructions contained within the REPEAT-UNTIL structure will be executed at least one time. If exp is true when it is evaluated, control is passed to the instruction following the UNTIL instruction. Otherwise program control is passed back to the REPEAT instruction.

Example:

```
1000 Input"Number, limit: ",Number,Limit
1010 Power'of'number=Number
1020 Repeat
1030 Print Power'of'number
1040 Power'of'number=Power'of'number*Number
1050 Until Power'of'number>=Limit
1060 End
```

This program will prompt the user for a number and a limit. The series of powers of the number will be displayed until the limit is reached.

statements: While-Endwhile Loop

```
format: Ln WHILE exp
        .
        .
        [program instructions]
        .
        .
        Ln ENDWHILE
```

where:

Ln are line numbers.

exp is an arithmetic or relational
 expression.

The WHILE structure is used to repeatedly execute a section of a Basic program WHILE a condition is true.

Notes:

1. If exp is false when it is evaluated, control passes to the instruction following the ENDWHILE instruction. Otherwise program control continues with the next sequential instruction.
2. It is possible for the code contained in the WHILE structure not to be executed at all. This will happen if exp is false the first time the WHILE instruction is executed.

12. Control Structures

Example:

```

110 Integer Valid'answer'flag,True,False
120 True=1 : False=0
130 Valid'answer'flag=False
140 While Not Valid'answer'flag
150 Input"Answer(y/n): ",Answer$
160 If Answer$="Y"Or Answer$="y"Then Valid'answer'flag=True
170 If Answer$="N"Or Answer$="n"Then Valid'answer'flag=True
180 If Not Valid'answer'flag Then Print"Invalid response. ";
190 Endwhile
200 Rem The program continues here,
210 Rem once a valid answer has been entered.

```

```

>>run
Answer(y/n): 57
Invalid response. Answer(y/n): new
Invalid response. Answer(y/n): Y

```

```

***END***

```

This section of code demonstrates the use of the WHILE structure. The WHILE condition (Not Valid'answer'flag) becomes false, and the program continues, only when the user provides an acceptable response.

The first program line defines the variables (Valid'answer'flag, True, and False) as type INTEGER. Since all Basic operations involving Integer variables are performed faster than those involving Floating Point variables, defining these variables as type INTEGER allows for faster program execution. Note that throughout the program, these are used strictly as Boolean-type variables. This means that they may only take on the values of true (=1) or false (=0). On the second line, the variables True and False are initialized.

On line 130, the variable Valid'answer'flag is set to False (=0) because, before the user is prompted, no valid answer has been accepted by the program.

Line 140 defines the condition for execution of the WHILE loop. This statement line can be read as, "WHILE the variable Valid'answer'flag is not true, perform the instructions up to the ENDWHILE instruction and then return control to this test (line 140)". If the portion of the statement following the WHILE is true, control remains within

12. Control Structures

the WHILE structure. If it is false, control is transferred to the instruction following the ENDWHILE. In this example, the boolean operator NOT causes control to remain within the WHILE loop as long as Valid'answer'flag is NOT true. When a valid answer is INPUT by the user, Valid'answer'flag is set to true, the WHILE test condition becomes false, and control passes to the instruction following the ENDWHILE.

Lines 150 through 180 prompt the user for a response and determine if the response meets the criteria of the programmer. If a valid response is INPUT, the value of the variable Valid'answer'flag is set to true (=1), otherwise it remains as false (=0). If Valid'answer'flag is false, an error message is displayed before the user is reprompted.

Upon encountering the ENDWHILE statement, control is transferred to the WHILE statement where a test is performed as indicated above.

12. Control Structures

statement: Stop Program Execution

format: Ln STOP

where:

Ln is a line number.

The STOP statement halts program execution and causes Basic to return to the command mode.

Notes:

1. After a program has been STOPped by a STOP statement execution may be restarted from the statement line immediately following the line containing the STOP statement by the use of the CONTinue command.
2. Upon execution of the STOP statement, Basic displays a message on the console indicating the line number of the STOP statement which caused the program to halt.

Example:

>>List

```

10 Input A,B,C,D,E,F
20 Let Number=A+B/C
30 Print A;B;C
40 Stop
50 Print D;E;F
60 Print Number
70 End

```

>>Run

? 1,2,3,4,5,6

123

40 Stop

>>Con

456

1.66666666666667

70 End

In this example, the Basic returns to command mode after encountering the STOP in statement line 40. The program is then CONTinued when the user enters the CON command.

13. Console and Data Input/Output

instruction: Input (from the console)

format: [Ln] INPUT var-1,var-2,...,var-n

[Ln] INPUT "string", var-1,var-2,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

var1-n is a list of one or more numeric and/or string variables.

string is an optional string literal. It is a prompt to be displayed on the console.

The INPUT instruction assigns a value, INPUT from the console, to a variable. When the INPUT instruction is executed, a prompt (either a question mark (?) or a string as used above) is output to the console. The user should respond to this prompt by entering a list of data which corresponds to the variable list in the INPUT instruction. The data list entered by the user must be terminated by a RETURN.

Notes:

1. If the string format of the INPUT instruction is used, the string will replace the question mark as the initial prompt.
2. If the user types in fewer data items than are called for in the variable list, a double question mark will appear on the terminal. This prompt continues to appear until each variable has been assigned a value. See the examples.
3. If more data is INPUT than is required, an error message will be generated.
4. The type of data INPUT must be the same as the type of variable listed in the INPUT instruction. For example, if the INPUT command contains a list of numeric variables,

13. Console and Data Input/Output

the data INPUT must be numeric data. If an attempt is made to INPUT string data into a numeric variable, an error message will be generated.

5. When a string variable is used with an INPUT instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to section 5.2.3, Referencing String Variables, for a complete discussion of the subject. The INPUT instruction will not move more characters than can be accepted by the destination string (or substring) which is being referenced.
6. If the INPUT list is terminated with a semicolon, the RETURN which the user types after the requested data has been entered will not be echoed. This allows more than one prompt and response sequence to appear on a single line.
7. Refer also to the description of the INPUT instruction in the chapter on Data File I/O.

Examples:

```
>>List
```

```
10 Input Num1,Num2,Num3,Num4
20 Print Num1
30 Input Num5,Num6
40 Print Num2;" ";Num3;" ";Num4;" ";Num5;" ";Num 6
```

```
>>Run
```

```
? 32,18,20,4
32
? 100,200
18 20 4 100 200
***50 End***
```

In the above example, the proper number of items was INPUT in response to each prompt. Suppose we were to respond to INPUT statement 10 with only three numbers:

13. Console and Data Input/Output

```
>>Run
? 5,10,15
?? 20
5
? 30,40
10 15 20 30 40
***50 End***
```

Basic displayed an additional prompt (??) when it did not receive as many items as were in the INPUT list.

In our example, line 10 and line 30 can be replaced with lines which will tell the user specifically what items are needed:

```
>>10 Input "Enter four numbers: ",Num1,Num2,Num3,Num4
>>30 Input "The last two numbers: ",Num6,Num7
>>Run
Enter four numbers: 100,200,300,400
100
The last two numbers: 500,600
200 300 400 500 600
***50 End***
```

instruction: Print (to the console)

format: [Ln] PRINT

[Ln] PRINT A1,A2,...,An

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

A1-n are numbers, string literals, numeric variables, string variables, numeric expressions, or standard or user defined functions.

The PRINT instruction is used to output information to the terminal.

Notes:

1. When used alone, the PRINT instruction will generate a new line (RETURN, LINE FEED).
2. The at sign (@) may be used in place of the word PRINT for brevity.

Example:

```
30 Print
40 Print "And it saves space"
```

can also be written as:

```
30 @
40 @ "And it saves space"
```

This feature facilitates entry of large, text-oriented programs.

3. The spacing of output can be controlled by using a comma (,) or a semicolon (;) between items in a PRINT list.

Items separated by commas are PRINTed beginning in the leftmost column of each PRINT

13. Console and Data Input/Output

field. Using the default value of 20 columns per field, the PRINT statement below:

```
10 Print 1,2,3,4
```

will produce output in the following format:

	1	2	3	4
column	0	20	40	60
no.				

See the SET instruction to change the number of columns per field.

If a semicolon is used between items, the items are printed adjacent to each other (i.e., without spaces between items). For example, the statement:

```
10 Print "Al"; "to"; "ge"; "th"; "er"
```

will produce output in the following format:

Altogether

Commas and semicolons may be used in a PRINT instruction in any combination.

4. If more items are listed in a PRINT statement than can be output on one line, Basic will generate a LINE FEED and continue PRINTing on the next line.
5. More complex formatting of PRINTed text is possible with the TAB and SPC functions and the PRINT USING instruction.
6. Refer also to the description of the PRINT instruction in the chapter on Data File I/O.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
13. Console and Data Input/Output

Example:

>>List

```
20 Current'year=1979
30 Print "This is ";Current'year
40 Print
50 Print
60 Input Birth'date
70 Age=Current'year-Birth'date
80 Print "You were born in ";Birth'date
90 Print
100 Print "You are "; Age;" years old this year"
110 End
```

>>Run
This is 1979

? 1952
You were born in 1952

You are 27 years old this year
110 End

13. Console and Data Input/Output

instruction: Read Data

format: [Ln] READ var-1,var-2,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

var1-n is a list of one or more numeric and/or string variables

The READ instruction is used to READ values from DATA statement(s) and assign these values to the READ instruction variable list.

Notes:

1. The order in which variables appear in the READ instruction determines which value from the DATA list will be assigned to which variable. For instance, the first value that appears in the DATA list is assigned to the first variable in the READ list, the fifth value is assigned to the fifth variable, and so forth.
2. A pointer is moved in sequence through the list of DATA values as these values are assigned to variables in the READ list. The number of DATA elements must be equal to or greater than the number of variables in the READ list. If there are fewer items remaining in the DATA list than are in the current READ list, an error message will be generated.
3. DATA elements corresponding to numeric variables must be numeric DATA and elements corresponding to string variables must be string literals.
4. When a string variable is used with a READ instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by

13. Console and Data Input/Output

the use of subscripts. Refer to section 5.2.3, Referencing String Variables, for a complete discussion of the subject. The READ instruction will not move more characters than can be accepted by the destination string (or substring) which is being referenced.

Example:

>>List

```
10 Read Bananas,Pears,Peaches,Company$
20 Print Bananas,Pears,Peaches,Company$
30 Data 10,20,30,"Fruit Co."
40 End
```

>>Run

```
10          20          30          Fruit Co.
***40 End***
```

instruction: Restore Data Pointer

format: [Ln] RESTORE

[Ln] RESTORE n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is a line number or line name in the current program.

The RESTORE instruction resets the DATA list pointer which is associated with the READ instruction. This allows the user to reREAD or skip over DATA items.

Notes:

1. Line n must be a line of or before a DATA statement. If n is not used, the DATA list pointer is positioned before the first DATA list item in the program. If n is included, the DATA list pointer is positioned before the first occurring DATA list item on or after the specified line.

13. Console and Data Input/Output

Example:

>>List

```
10 Read A,B,C,D
20 Read E,F,G,H
30 Restore
40 Read R,S,T,U
50 Restore 90
60 Read L,M,N,O
70 Data 1,2,3,4
80 Data 5,6,7,8
90 Data 9,10,11,12
100 Print A,B,C,D
110 Print E,F,G,H
120 Print R,S,T,U
130 Print L,M,N,O
```

>>Run

1	2	3	4
5	6	7	8
1	2	3	4
9	10	11	12

End

13. Console and Data Input/Output

statement: Data

format: Ln DATA A1, A2,...,An

where:

Ln is a line number.

A1-An are constants, string literals, numeric variables, string variables, numeric expressions, or standard or user defined functions.

The DATA statement specifies values for variables appearing in a READ instruction.

Notes:

1. The READ instruction is used to READ values from DATA statements and assign these values to the READ instruction variable list.
2. The RESTORE instruction allows items in the DATA list to be skipped or reREAD by resetting the DATA list pointer.

Example:

10 Data 7*3,9+(2*107),"This is Data",15,75

14. Output Formatting

instruction: Print Using

format: [Ln] PRINT USING svar, A1,A2,...,An

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string literal or string variable format definition.

A1-n are constants, string literals, numeric variables, string variables, numeric expressions, or standard or user defined functions.

The PRINT USING instruction allows the user to specify a format for displaying output.

Notes:

In examples in this section the character b represents a blank character.

1. A format field is bounded on either side by any character which is not one of the special characters (# & * + , \$! - .). As indicated in the general format for the PRINT USING instruction, a format expression may include more than one format field and may also include string literals. If multiple format fields are specified, the values of expressions are assigned to these fields in order. A format expression may also be assigned to a string variable.

Example:

>>List

```
10 Num1=11 : Num2=333 : Num3=22
20 Dim Format$(20) : Format$="### &&&M&&&"
30 Print Using"### &&&M&&&",Num1,Num2,Num3
40 @Using Format$,Num1,Num2,Num3
50 End
```

14. Output Formatting

```
>>Run
  11 0333M022
  11 0333M022
***50 End***
```

Statement 30 in the above example outputs the three variables (Num1, Num2, and Num3) with 2 spaces separating the first two and the literal M separating the last two. Any literal which is not one of the PRINT USING special characters may be used to separate format fields. This literal will be PRINTed between the fields if it is a blank or any other character. Statement 40 outputs the same information in the same format using a string variable instead of a string literal in the PRINT USING instruction. Statement 40 also abbreviates PRINT with the symbol @.

2. When a string appears in a PRINT USING list, the characters of the string are PRINTed in the positions held by any of the special format field characters. Strings are left justified in the format field. If the number of characters in the string is less than the number of characters in the format field, the extra spaces will be blank filled. If the number of characters in the string is greater than the number of characters in the format field, the extra characters in the string will be truncated.

Examples:

In the instruction:

```
Print Using "****,**.\"", "ABCDEF"
the string literal is output in the format:
ABCDEFbbb
```

In the instruction:

```
Print Using "&&&,&&&.&&\"", "ABCDEFGH IJKLMN"
the string literal is output in the format:
ABCDEFGH IJ
```

3. If the number of items in the expression list exceeds the number of specified format fields, the specified format fields will be re-used for the extra items.

14. Output Formatting

Examples:

In the instruction:

Print Using "\$\$&&.&&", A,B,C
the expressions A, B, and C will all be
PRINTed with the format field \$\$&&.&&.

In the instruction:

Print Using "\$\$##bb\$\$&&.&", A,B,C
the expressions A and C will be formatted
using the format field \$\$## and the expression
B will be formatted using the format field
\$\$&&.&.

4. All normal PRINT functions (such as TAB, SPC, semicolon, and comma) are overridden by the PRINT USING instruction.
5. The format expression may include a maximum of 128 characters.
6. If a number being formatted has more digits than allowed for in the format expression, an all asterisk error message will result.

Digit Formatting

These special characters may be used to format digits:

#	indicates leading blanks
&	indicates leading zeroes
*	indicates leading asterisks

These symbols are used to right justify digits in a PRINT field. The width of this PRINT field is determined by the number of special characters included in a format field. Any non-digits (such as a minus sign) are eliminated. If the number of special characters in the format field exceeds the number of digits in the expression, the digits will be right justified within the field and preceded by characters corresponding to the special characters used in the format field. In the following examples, the character b is used to represent blank characters (spaces).

14. Output Formatting

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
1	#####	bbbb1
12	#####	bbb12
123	#####	bb123
1234	#####	b1234
12345	#####	12345
123456	#####	*****
1	&&&&&	00001
12	&&&&&	00012
123	&&&&&	00123
1234	&&&&&	01234
12345	&&&&&	12345
123456	&&&&&	*****
1	*****	****1
12	*****	***12
123	*****	**123
1234	*****	*1234
12345	*****	12345
123456	*****	*****

Comma (,)

The comma (,) places a comma in the position in which it appears in the format field. If the format specifies that a comma be output in a position in the field which consists of leading blanks, zeroes, or asterisks, then a blank, a zero, or an asterisk respectively are printed in the comma position.

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
2003	##,###	b2,003
4	##,###	bbbbbb4
4457	&&&,&&	044,57
18	&&&,&&	000018
996546	*****,*	99654,6
22	*****,*	****2,2

14. Output Formatting

Decimal Point (.)

The decimal point (.) places a decimal point in the position in which it appears in the format field. All digit positions which follow the decimal point are filled with digits. If the expression contains fewer fractional digits than are specified, zeroes will be PRINTed in the extra positions. If the expression contains more fractional digits than are specified, the expression will be rounded so that the number of fractional digits equals the number of format positions specified.

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
234	###.###	234.000
23.4567	###.###	b23.457
13	&&.&	13.0
66.72319	&&.&	66.7
876.1245	*****. **	**876.12
1234567.245	*****. **	*****. **

In the last example, when too many significant digits appear to the left of a decimal point, an all asterisk error message is generated.

Fixed Plus (+) and Minus (-) Signs

The plus (+) and minus (-) signs may appear in the first character position in a format field. The character + or - will print the respective sign of an expression in the specified character position in the format field. When an expression is preceded by a plus or minus sign, any leading zeroes will be replaced by blanks, zeroes, or asterisks as specified. When a positive expression is preceded by a minus sign, a blank space is left in the sign position.

14. Output Formatting

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
56.8888	+###.###	+56.889
4.564	+###.###	+b4.564
6.456	+&&&.&&	+006.46
234.2	+&&&.&&	+234.20
-23.56	-*****.*	-***23.6
2345	-*****.*	b*2345.0
-2345678.34	-*****.*	*****.*

Floating Plus (++) or Minus (--) Signs

The use of two or more plus or minus signs at the beginning of a format field will output the respective sign directly preceding the value of the expression. If a positive expression is PRINTED USING a floating minus format, a blank is PRINTED immediately preceding the number instead of a minus sign. The additional signs in the floating point format can be used to represent digits.

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
2.234	++###.###	b+b2.234
22.234	++###.###	b+22.234
-44.56	--&&&.&	b-044.6
5.32	--&&&.&	bb005.3
178.456	++*****.*	b+*178.46
12345678.45	++*****.*	*****.*
55.17	+++++.++	bb+55.17
55.17	-----.--	bbb55.17
-55.17	-----.--	bb-55.17

Fixed Dollar Sign (\$)

The dollar sign (\$) is used in either the first or second character position in the format field to PRINT out a dollar sign in that position. A dollar sign specified in the second position of the format field must be preceded by either a plus (+) or a

14. Output Formatting

minus (-) sign.

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
23.456	\$##.##	\$23.46
4.52	\$##.##	\$b4.52
-57.654	-\$&&&.&&&	-\$057.654
123.7789	-\$&&&.&&&	b\$123.779
2.34	\$*.*	\$2.3
234.55	\$*.*	**.*

Floating Dollar Sign (\$\$)

The use of two or more dollar signs beginning at either the first or second character position in the format field will output a dollar sign immediately preceding the value of an expression. If the dollar signs begin in the second character position, the first character position must contain a plus or a minus sign.

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
234.2345	\$\$\$\$#.###	b\$234.235
4.45	\$\$\$\$#.###	bbb\$4.450
23.989	\$&&&.&&	b\$23.99
4.5	\$&&&.&&	b\$04.50
24.56	-\$****.**	bb\$*24.56
-4455.67	-\$****.**	-\$4455.67

Exponent Fields (!!!!)

Four consecutive exclamation characters (!!!!) indicate an exponent in the format field. The exclamation points represent the expression E+nn, where n is any digit. When used with a numeric expression, this format field will output the expression in exponential form.

14. Output Formatting

Examples:

<u>Value of Expression</u>	<u>Format Field</u>	<u>Output</u>
23.3456	##.##!!!!	23.35E+00
2000	##.##!!!!	20.00E+02
-.36	-&&.&&!!!!	-360.00E-03

7. Only the characters #, &, or ! should be used to the right of a decimal point. The asterisk (*) character follows the same rules as the # character when used to the right of a decimal point.
8. Only one type of floating character may be used within a single format instruction. Either the floating dollar (\$\$) character, pluses (++), or minuses (--) may be used, but may not be combined.
9. A non-floating PRINT character cannot be placed to the left of a floating character. For example, the format fields \$+++ or +\$\$\$ are legal but the format field +\$++ is illegal.
10. Structured Basic does not check comma syntax.
11. Only one decimal point may be used in a format field.
12. Trailing + or - signs are illegal.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
14. Output Formatting

function: Space

format: SPC(aexp)

where:

aexp is an arithmetic expression,
variable, or constant.

The SPC function instructs Basic to PRINT a specified number of SPaCes.

Notes:

1. The SPC function may be used only in conjunction with a PRINT instruction. If used elsewhere, it is a transparent function:

Value = Spc(1.47)

is the same as

Value = 1.47

2. Unlike the TAB function, which always determines PRINT position relative to column 0, the SPC function determines PRINT position relative to the current PRINT column.

Example:

>>List

```
10 Num1=2
20 Input Num2,Num3
30 Print Spc(10);Num1;Spc (Num1*10);Num2;Num3
40 End
```

>>Run

? 20,30

2

2030

column no. 10

31

In this example, the computer is instructed to skip 10 SPaCes, PRINT Num1, skip 20 SPaCes, and PRINT Num2 and Num3.

14. Output Formatting

function: Tab

format: TAB(aexp)

where:

aexp is an arithmetic expression,
variable, or constant.

The TAB function causes output to begin in a specified column.

Notes:

1. The TAB function may be used only in conjunction with a PRINT instruction. If used elsewhere, it is a transparent function:

Value = Tab(6.2)

is the same as

Value = 6.2

2. Multiple TAB functions may be included in one PRINT instruction, but the user should keep in mind that the PRINT position indicated by successive TAB functions is always determined relative to column 0.
3. Columns are numbered 0 through the page width so the first column is column 0.
4. If the argument to the TAB function exceeds the current page width, it is reduced modulo that page width to a number between 0 and the page width. The default value for the page width is 80 characters.
5. If the argument is negative, no TABbing takes place.
6. If the (reduced) argument is greater than the current column position, a new line (RETURN-LINE FEED) is issued and the TAB is executed on the next line.

14. Output Formatting

Example:

>>List

```
10 Addtab=1
20 Input Num1,Num2,Num3
30 Print Tab(2);Num1;Tab(5);Num2;Tab(Addtab+9);Num3
40 End
```

>>Run

? 5,8,9

5 8 9

40 End

In this example, the computer is instructed to begin PRINTing Num1 in column 2, Num2 in column 5, and Num3 in column 10.

15. Discussion - Data File Input/Output

Data File Input/Output

15.1 Data Files

Data is information. A file is a place and method for storing many individual items of information.

A computer data file (or file) is defined by:

1. The storage medium (paper tape, floppy disk, etc.),
2. The method of accessing the data (sequential or random),
3. The code by which the data is translated for storage (ASCII or internal machine representation).

15.1.1 Records

A record is a group of related items. A data file is made up of records. Information is inserted into or retrieved from the file record by record.

If a file contained information on all of the baseball games in one year, each record might contain information on one game, one player, or one team. All of the records would contain similar information. If the first record (record number 0) held the statistics on game #1, the first number in the first record could be the number of hits team A got, while the second number would be the number of hits team B got. The third number could be the number of errors made by team A, while the fourth number would be the number of errors made by team B. The second record (record number 1) would contain similar information covering game #2.

15. Discussion - Data File Input/Output

15.1.2 Fields

A field describes one item in a record. A field can contain string (alphabetic) or numeric information.

In the above example, instead of saying that the first number was the number of hits team A got, we could say that the first field was numeric and contained the number of hits team A got. We can now further describe the record layout by saying that the fifth and sixth fields are alphabetic and contain the team names.

15.2 Floppy Diskette

A floppy diskette (or disk) is a flat, round magnetic storage medium. It is protected by a square envelope which exposes only a small portion of the surface of the disk in addition to the hole in the middle. When used in conjunction with a disk drive and disk controller, the disk becomes a data file storage device. Data can be added to the data already on the disk or records on the disk can be changed or deleted.

15.3 Creating a Data File

Before the Cromemco Disk Operating System (CDOS - the operating system through which all disk operations are performed, even if the programmer is using Basic) can OPEN a disk file, it must CREATE the file in the file directory.

A file can be CREATED from CDOS by using the CDOS Text or Screen Editor. A file can also be CREATED from Basic by using the CREATE instruction.

15. Discussion - Data File Input/Output

15.3.1 OPENing an I/O Channel

Once a file has been CREATED, it must be OPENed to allow Basic access to the file. The OPEN instruction reserves an Input/Output channel and assigns the I/O channel number to the file reference or file device. Once a file has been OPENed (and assigned a channel number) all input and output from/to that file will be done through the assigned channel.

Basic normally carries 8 I/O channels in addition to the console and the Procedure Library. One I/O channel is needed for each file which is OPENed at the same time. Each channel occupies 192 bytes of memory. Increasing the number of channels allows more files to be OPENed simultaneously but reduces the amount of memory available to the Basic user. Conversely, decreasing the number of I/O channels increases the amount of available memory. If you wish to change the number of channels available for use, see the Appendix, Changing the Number of I/O Channels.

15.3.2 CLOSE

The CLOSE instruction will disassociate the Input/Output channel number and the file which were associated by the OPEN instruction. The channel will then be available for use by other files.

15.4 Internal Machine vs. ASCII Representation

Basic can store data in two formats. Both consist of groups of ones and zeros (binary representation), but each needs to be translated differently for output to a terminal or printer.

ASCII (American Standard Code for Information Interchange) is the closest to written letters and numbers. ASCII is stored in the machine according to the table of ASCII codes (see the Appendix). Each character code occupies one byte (8 bits). When the code is translated from binary to decimal, the character which is represented can be found in the ASCII table. This is the way all Basic strings are stored. ASCII information is stored so that

15. Discussion - Data File Input/Output

one character is stored in one byte.

The need arises for another method of storing numbers when speed of arithmetic computation, Input/Output translation, and amount of storage space become important factors.

Internal Machine code is used only for the storage of numbers. The number of bytes occupied by a number depends on its magnitude and precision (the size of the number and the number of decimal places which need to be kept.) There are three internal machine formats:

Integer	size: 2 bytes range: $+32767 < N < -32768$ accuracy: nearest integer
Short Floating Point	size: 4 bytes range: $+9.99E+62 < N < +9.99E-65$ accuracy: <u>6</u> decimal digits
Long Floating Point	size: 8 bytes range: $+9.99E+62 < N < +9.99E-65$ accuracy: <u>14</u> decimal digits

Refer to Chapter 3 for a more complete discussion of internal machine representation.

15.5 PRINT and INPUT

PRINT and INPUT write and read in ASCII format. These instructions are primarily intended to write data to an output file which is to be PRINTED or Typed (using CDOS) at a later time, and to read a data file which was created by the Screen or Text Editor or another system. When used with a file, they exactly parallel their operation with the console terminal. In particular, file INPUT requires a comma or RETURN between INPUT data items.

It is recommended that the PUT and GET instructions be used with data files which are to be accessed by Basic programs.

The PRINT and INPUT instructions translate numeric data from ASCII to Internal Machine Format (or vice versa). They are therefore slower than the PUT and

15. Discussion - Data File Input/Output

GET instructions. There is little need for the programmer to keep track of the type of variable which is written out with a PRINT instruction. An integer variable which is written using the PRINT instruction may be INPUT back into a string or floating point variable without losing its meaning. No attempt should be made to INPUT an ASCII string into a numeric variable. A file which has been output using the PRINT instruction may also be Typed out (using CDOS) and read whereas the same file output with the PUT instruction would not be able to be Typed.

15.6 PUT and GET

The PUT and GET instructions write and read in internal machine format. If an integer variable is written out to a file using a PUT instruction, it must be read back into an integer variable (and not into a floating point variable or an ASCII string) using a GET instruction or it will be meaningless. The machine does no translation with these instructions so they can be executed faster than PRINT and INPUT. The programmer must keep track of the types and lengths of variables written by PUT so that they may be read back in properly.

15.7 An Interesting Note

A string (which is stored internally in ASCII code) may be output with a PUT or PRINT instruction with the same general result (except that PRINT outputs a carriage RETURN-LINE FEED sequence where appropriate and tabs, using spaces, when variables are separated by commas). This is because PUT will output a string in internal machine format, which (for a string) is ASCII. The PRINT will not translate the string because it is already in ASCII format.

15. Discussion - Data File Input/Output

15.8 File Pointer

While a file is OPEN, Basic maintains a pointer which may determine where the next read or write will occur. This pointer can be manipulated by the various file read and write instructions as will be explained in the following paragraphs.

When a file is first OPENed, the File Pointer is positioned just before the first byte of the first record of the file. If the file is a new file (i.e., it contains no data), the beginning of the file and the end of the file coincide, and so the File Pointer also points to the end of file. This is appropriate. If a read is attempted on a new file, an end of file error will be returned because there is no data in the file.

If no record number is specified in a file I/O instruction (sequential I/O), the data list contained in the instruction is written to or read from the file from the current position of the file pointer.

If a record number is specified in a file I/O instruction (random I/O), the File Pointer is moved to a position just before byte zero of the specified record and then the data list contained in the instruction is written to or read from the file from the new position of the File Pointer.

If a byte number is specified in addition to the record number, the File Pointer will be positioned just before the specified byte in the current or specified record before the I/O instruction is executed.

If there is no data list associated with a PUT instruction, the pointer is repositioned as specified above and no input or output takes place.

After the input or output has taken place, the PUT and GET instructions leave the File Pointer just after the byte which was last input or output.

The Print and Input instructions use the Carriage RETURN or Carriage RETURN-LINE FEED sequence as a delimiter.

The PRINT instruction will move the File Pointer to a position just after the last PRINT character. If

15. Discussion - Data File Input/Output

the PRINT line starts with byte zero of the current record and is the same length as the specified record size, the pointer will be positioned just before byte zero of the next logical (sequential) record.

The PRINT instruction will automatically insert a RETURN-LINE FEED sequence if:

1. the data list associated with the Print instruction will cause a record to be output which is longer than the page width (refer to the SET instruction), and
2. no single item in the data list is longer than the specified page width.

This can happen when items in a PRINT list are separated by commas or a PRINT instruction is terminated with a comma or semicolon and a subsequent PRINT instruction adds to the already PRINTed line. This rule also applies to the PRINT instruction when used to send output to the terminal.

A PRINT list containing an item which is longer than the page width will cause a run time error. The default page width is 80 characters. It is possible to change this parameter by using the SET instruction.

15.9 Sequential Files

A sequential file is written in the order of the record numbers. That is, record number zero is written with the first output (PUT or PRINT) instruction, record number one is written next, then record number two, etc. This continues until the file is CLOSED. When the file is OPENed again, the first record which is output will write over record number zero, etc.

When a file is read sequentially, record zero is read first, then record one, etc.

Note that if an entire record is not input or output each time the file is accessed, the File Pointer will remain after the last character which was read or written. Any subsequent file access

15. Discussion - Data File Input/Output

will read or write from that position, not from the beginning of the next record.

15.10 Random Files

A random access file is written in the order specified by the programmer. Each output instruction must specify the number of the record which is to be written. When a file is read randomly, the programmer must specify the record number to be read for each INPUT or GET instruction.

A file which was written sequentially may be read as a random file. Files which were written randomly may be read by sequential or random instructions provided that no attempt is made to read a record before it has been written.

A file may be accessed by any combination of sequential and random instructions.

A random instruction, one that specifies a record or record and byte number, will reposition the File Pointer before accessing the file.

A sequential instruction, one that does not specify a record number, will access the file from the current position of the File Pointer.

15.11 CDOS Dump Utility

The CDOS Dump utility program is very useful for determining exactly what has been sent to a file. The user must exit from Basic (using BYE) in order to use this program. Refer to the CDOS manual, Utility Programs - Dump, for further information.

instruction: Create File

format: [Ln] CREATE file-ref

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref includes the 1 to 8 character file name and optionally a disk drive specifier and file name extension. This may be a string literal enclosed in quotation marks or a string variable.

The CREATE instruction places the file name (and file name extension, if one is used) in the directory of the specified disk. A file may only be CREATED once, and must be CREATED before it can be OPENED.

Notes:

1. The file reference is composed of the disk drive specifier and/or the file name and/or the file name extension. These may be a string variable or a string literal enclosed in quotation marks.
2. The file name and file name extension may include any printable ASCII character except the following:

\$ * ? = / . , : - "space"
3. No space is allocated to a file by the CREATE instruction. All file space under CDOS is dynamically allocated only when needed.
4. Error Number 137 (File Already Exists) will result if the file already exists in the directory when the CREATE instruction is given.

15. Instructions - Data File Input/Output

instruction: Open File

format: [Ln] OPEN\n\ file-ref

[Ln] OPEN\n,p1\ file-ref

[Ln] OPEN\n,p1,p2\ file-ref

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is the required file (channel) number.

p1 (for a disk file) is the optional record size in bytes which may be assigned any value between 1 and 32,767. The default value is 128 bytes per record (one sector).

p2 (for a disk file) is the optional file access mode specifier:

=1 is read only
 =2 is write only
 =3 is read and write
 (default value)

file-ref includes the 1 to 8 character file name and optionally a disk drive specifier and file name extension. This may be a string literal enclosed in quotation marks or a string variable.

The OPEN instruction allows a disk file or system device to be linked to a file number (channel) for future reference in connection with file Input/Output instructions (i.e., PUT, GET, INPUT, PRINT, CLOSE).

Notes:

1. The file reference is composed of the disk

15. Instructions - Data File Input/Output

drive specifier and/or the file name and/or the file name extension. These may be a string variable or a string literal enclosed in quotation marks.

2. The file name and file name extension may include any printable ASCII character except the following:

\$ * ? = / . , : - "space"

3. The file number must be between 1 and the maximum channel number available with an absolute maximum of 16. As Basic is shipped, the maximum channel number is 8. See the Appendix for the method of changing the number of channels available.
4. Error number 133 (File Number) will result if an attempt is made to reference a file number greater than the maximum channel number available.
5. The file number 0 (zero) is reserved for the console. It cannot be OPENed by the user. All input/output (GET, INPUT, PUT, and PRINT) directed from/to file 0 will use the console.

Caution: If file number 0 is specified by the CLOSE instruction, all currently OPENed files will be CLOSED.

6. Although files OPENed for read/write access may be used as write only files, slightly faster execution speed may result if the file is OPENed for write only access.
7. If the PRINT instruction is used to write a single item which is longer than the current page width, it will be necessary to increase the page width (using the SET instruction) in order to avoid Error number 6 (Print Item Size). Refer to the discussion section of this chapter for additional information.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
15. Instructions - Data File Input/Output

instruction: Close File

format: [Ln] CLOSE

[Ln] CLOSE \n\

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is an optional file (channel) number. The default value is all currently OPENed files.

The CLOSE instruction disassociates the channel number and file which were associated by the OPEN instruction.

Note:

1. If the CLOSE instruction is given without a file number or with the file number set equal to zero, all currently OPENed files will be CLOSED.

15. Instructions - Data File Input/Output

instruction: Erase File

format: [Ln] ERASE file-ref

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref includes the 1 to 8 character file name and optionally a disk drive specifier and file name extension. This may be a string literal enclosed in quotation marks or a string variable.

The ERASE instruction will remove a file from the file directory.

Examples:

```
Erase "TEMP.BAS"  
Erase "*.BAK"  
Erase "B:TEMP.REL"
```

The first of these examples will ERASE the file named TEMP.BAS from the current or default disk drive. The second will ERASE all files from the current disk drive with the file name extension of BAK. The third example will ERASE the file called TEMP.REL from the disk in the B drive.

15. Instructions - Data File Input/Output

instruction: Rename File

format: [Ln] RENAME file-ref-1, file-ref-2

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

file-ref-1 is a file reference to the existing file which includes the 1 to 8 character file name and optionally a disk drive specifier and file name extension. This may be a string literal enclosed in quotation marks or a string variable.

file-ref-2 is a file reference to the new file which includes the 1 to 8 character file name and optionally a disk drive specifier and file name extension. This may be a string literal enclosed in quotation marks or a string variable.

The RENAME instruction is used to give a new file name to a file already in the file directory.

Example:

Rename "Oldfile", "Newfile"

This command will change the name of the file (on the current disk drive) from OLDFILE to NEWFILE.

15. Instructions - Data File Input/Output

instruction: Print

format: [Ln] PRINT

[Ln] PRINT exp-1,...,exp-n

[Ln] PRINT\n\

[Ln] PRINT\n,p1\

[Ln] PRINT\n,p1,p2\

[Ln] PRINT\n\ exp-1,...,exp-n

[Ln] PRINT\n,p1\ exp,...,exp-n

[Ln] PRINT\n,p1,p2\ exp-1,...,exp-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is an optional file number. If no file number is specified, or if the file number is zero, output goes to the console terminal.

p1 (for a disk file) is the optional record number. The default value is sequential access starting with record 0 or the current position of the File Pointer.

p2 (for a disk file) is the optional byte number. The default value is byte 0.

expl-n is an optional list of expressions, numeric or string variables, or string literals. The data list must be separated by either semicolons or commas. If an item is followed by a comma, the item following it will be printed starting in the next tab position; if it is

15. Instructions - Data File Input/Output

followed by a semicolon no space will be left before printing the next item.

When used without a file reference, the PRINT instruction will cause the data list to be output to the console. When used with a file reference, it will cause the data list to be output to an ASCII device (e.g., the line printer) or a disk file in ASCII format.

Notes:

It is recommended that PRINT only be used for the output of PRINT or list files and not be used for writing data files which are to be read back by a Basic program.

1. The at (@) sign may be used to abbreviate the word PRINT in this instruction.
2. Most of the above forms of the PRINT instruction may incorporate the PRINT USING feature. Refer to PRINT USING for additional information.
3. The PRINT instruction outputs a RETURN-LINE FEED (ASCII 13,10) sequence at the end of the data list. If the file is to be read in using the INPUT instruction, a RETURN-LINE FEED sequence (or just a RETURN) must follow each item which is output, as it is this character or character sequence which delimits (terminates) each item. In Basic, there are two ways this can be accomplished:
 - a. Allow only one item per PRINT instruction.
 - b. Insert a RETURN between each item as follows:

```
100 Cr$ = CHR$(13)
200 Print \1\ "String"; Cr$; Num; Cr$; Text$
```

Statement 100 assigns the ASCII value of the RETURN to the string variable Cr\$. Statement 200 PRINTs a string literal, a numeric variable, and a string variable, each separated by the RETURN.

15. Instructions - Data File Input/Output

The above procedure will allow all of the items output by the PRINT instruction to be read by the INPUT instruction which looks for a RETURN between each item read.

If only numeric data is to be PRINTed and INPUT a comma (ASCII 44) may be used as a delimiter between variables within one INPUT list. The last item in the data list must be delimited by a RETURN.

4. Specifying a negative number for either p1 or p2 will result in the default value being assigned to that parameter.
5. The PRINT instruction, when given without a data list, causes one blank line to be sent to the console or PRINT file.

15. Instructions - Data File Input/Output

instruction: Input

format: [Ln] INPUT var-1,...,var-n

[Ln] INPUT\n\ var-1,...,var-n

[Ln] INPUT\n,p1\ var,...,var-n

[Ln] INPUT\n,p1,p2\ var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is an optional file number. If no file number is specified, or if the file number is zero, input is received from the console terminal.

p1 (for a disk file) is the optional record number. The default value is sequential access starting with record 0 or the current position of the File Pointer.

p2 (for a disk file) is the optional byte number. The default value is byte 0.

var1-n is a list of one or more numeric and/or string variables.

When used without a file reference, the INPUT instruction will cause the data list to be INPUT from the console. When used with a file reference, it will cause the list to be INPUT from an ASCII device (e.g., paper tape reader) or a disk file in ASCII format.

Notes:

1. When a string variable is used with an INPUT instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string

15. Instructions - Data File Input/Output

variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to section 5.2.3, Referencing String Variables, for a complete discussion of the subject. The INPUT instruction will not move more characters than can be accepted by the destination string or substring which is being referenced.

2. INPUT can accept no more than 132 characters per line.
3. INPUT retrieves only 7 bit ASCII. The GET instruction must be used to retrieve all 8 bits.
4. INPUT treats some control characters as editing or end of file commands.

15. Instructions - Data File Input/Output

instruction: Put Record

format: [Ln] PUT\n\

[Ln] PUT\n\ exp-1,...,exp-n

[Ln] PUT\n,p1\ exp,...,exp-n

[Ln] PUT\n,p1,p2\ exp-1,...,exp-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is a file number. If file number 0 is used, output will be set to the console.

p1 (for a disk file) is the optional record number. The default value is sequential access starting with record 0 or the current position of the File Pointer.

p2 (for a disk file) is the optional byte number. The default value is byte 0.

expl-n is a list of one or more numeric and/or string variables.

The PUT instruction outputs the data in the data list (exp-1,...exp-n) to the file specified by the file number (n). The data is output in internal machine format. This is useful if the data is to be read back by Basic. The PRINT instruction should be used if the file is to be Typed or LISTed to the printer.

Notes:

1. If no output list is included, only the device status is set (i.e., record and byte position on a disk file). This is a useful way of setting status (position) without actually initiating any data transfer.

15. Instructions - Data File Input/Output

2. The PUT instruction will output data to a file in internal machine format (refer to the discussion at the beginning of this chapter on Internal Machine vs. ASCII Representation). Numeric data which has been output using the PUT instruction must be read back in using the GET instruction.
3. ASCII data which is output by the PUT instruction will not be readable by INPUT unless a RETURN appears at least every 132 bytes.

15. Instructions - Data File Input/Output

instruction: Get Record

format: [Ln] GET\n\ exp-1,...,exp-n

[Ln] GET\n,p1\ exp-1,...,exp-n

[Ln] GET\n,p1,p2\ exp-1,...,exp-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is a file number. If file number 0 is used, input will be accepted from the console.

p1 (for a disk file) is the optional record number. The default value is sequential access starting with record 0.

p2 (for a disk file) is the optional byte number. The default value is byte 0.

exp1-n is a list of one or more numeric and/or string variables.

The GET instruction GETs the data in the data list (exp-1,...exp-n) from the file specified by the file number (n). The data is input without translation (internal machine format). This is useful if the data has been output by the PUT instruction.

Note:

1. When a string variable is used with a GET instruction, the portion of the string which is referenced is set equal to null characters before the source is moved into the string variable. The whole string is referenced if no subscripts follow the string variable, while various substrings may be referenced by the use of subscripts. Refer to section 5.2.3, Referencing String Variables, for a complete discussion of the subject. The GET instruction will not move more characters than

15. Instructions - Data File Input/Output

can be accepted by the destination string or substring which is being referenced.

15. Instructions - Data File Input/Output

Example Program - Random Access Files

```

10  Rem This is a program which demonstrates
20  Rem the implementation of a Random
30  Rem Access File.
40  Rem
50  Rem The program creates and opens a file, writes
60  Rem 51 records sequentially, allows
70  Rem the user to repeatedly select any
80  Rem of the 51 records at random, and
90  Rem then closes and erases the file.
100 Rem
110 Dim A$(21)
120 Integer Index,Number
130 Create"Randtest"
140 Open\1,24\"Randtest"
150 On Error Goto Query
160 On Esc Goto Finish
170 Print"Pause, writing file Randtest"
180   For Index=0 To 50
190     Put\1\"This is record number ",Index
200   Next Index
210 *Query : Print
220   Print"Which record would you like to see?"
230   Print"Enter record number (0-50) or -1 to stop: ";
240   Input Index
250   If Index<0 Then Goto Finish
260   If Index>50 Then Goto Query
270   Get\1,Index\Text$(-1),Number
280   Print"Contents of record number ";Index;" is:"
290   Print Text$;Number
300   Print
310   Goto Query
320 *Finish : Close
330   Erase"Randtest"
340   Print : Print : Print"File Randtest erased"
350   End

```

On line 130, the file Randtest is CREATED. If a file does not exist in the directory, it must be CREATED before it can be OPENED for reading or writing. The file is OPENED on line 140 with a record length of 24 bytes which are allocated as follows:

22 bytes for the string, and
2 bytes for an integer

Line 150 ensures that any run time error (such as an invalid user response) will return control to

15. Instructions - Data File Input/Output

the user within the program and will not cause the program to terminate abnormally. Because many programs can be terminated by the use of the ESCape key, statement 160 is included in this program. The program will still be terminated by depressing the ESCape key, but the active file will be closed and deleted before control is returned to the user.

Statements 180 through 200 write out 51 records, each containing (for identification) the record number in addition to a string. Because no record numbers are specified, the records are written sequentially starting with record number zero.

The user is then asked for the number of the record to be retrieved or displayed and on line 270 the record whose number is specified by the variable I is retrieved. Because the records which were written to this file each contained a 22 character string followed by an integer number, they must be read back into variables of the same type and length.

The user is allowed to view as many records as desired. Entering a negative one (-1) when asked for a record number will cause the file to be CLOSED and ERASEd and program execution to be terminated.

Refer to the discussion section of this chapter for more information on the use of files.

FUNCTIONS

Cromemco Structured Basic includes a number of arithmetic and trigonometric functions which perform common, frequently used calculations. These functions are pre-defined in Basic so that the programmer does not need to write a program everytime one of these functions is required.

Cromemco Basic also includes a number of functions designed to increase string handling capabilities, system functions which provide general system information, and functions which makes it possible to call assembly language subroutines.

In addition to these pre-defined functions, Cromemco Basic permits the programmer to define additional functions.

Cromemco Structured Basic includes the following functions:

Arithmetic Functions

Abs(X)	absolute value of X
Binadd(X,Y)	binary addition
Binand(X,Y)	binary logical And
Binor(X,Y)	binary logical Or
Binsub(X,Y)	binary subtraction
Binxor(X,Y)	binary logical Exclusive Or
Exp(X)	"e" to the power X
Fra(X)	fractional portion of X
Int(X)	integer value of X

16. Functions

IrN(X)	generates an integer random number between 0 and 32767
Log(X)	natural logarithm of X
Max(X1,..,Xn)	returns the numeric expression Xn with the maximum value in the expression list
Min(X1,..,Xn)	returns the numeric expression Xn with the minimum value in the expression list
Randomize	used with Rnd and Irn to produce different sets of random numbers
Rnd(X)	generates a random number between 0 and 1
Sgn(X)	algebraic sign of X
Sqr(X)	square root of X

Trigonometric Functions

Atn(X)	arctangent of X
Cos(X)	cosine of X
Sin(X)	sine of X
Tan(X)	tangent of X

Programmer Defined Functions

Def Fns(X1,...,Xn) = Y
 allows the user to define different functions

String Functions

Asc(X\$)	provides equivalent ASCII numeric value of the first character of X\$
Chr\$(X)	gives a single character string which is the ASCII equivalent of X
Expand X\$,n	inserts n null characters in string X\$
Hex\$(X)	returns the 4 byte ASCII hexadecimal representation of a number.
Len(X\$)	returns the length of string X\$.
Pos(X\$,Y\$,n)	returns the location of substring Y\$ within string X\$ starting with character n.
Str\$(X)	returns the character representation of any numeric expression X.
Val(X\$)	returns the numeric representation of any string expression X\$.
Valc(X\$)	returns the numeric representation of any string expression X\$. Sets an error condition for unacceptable values of X\$.

Descriptions and examples of each of these functions are included in the following section.

Also see Chapters 17 and 18 for descriptions of additional functions.

16.1 Arithmetic Functions

function: Absolute Value

format: ABS(aexp)

where:

aexp is an arithmetic expression,
variable, or constant.

The ABS function gives the ABSolute (i.e., positive) value of aexp, which can be any arithmetic expression.

Example:

>>List

```
10 Print Abs(-26), Abs(26)
20 End
```

>>Run

```
26
***20 End***
```

16.1 Arithmetic Functions

functions: Binary Operations

format: BINADD(aexp-1,aexp-2)

BINAND(aexp-1,aexp-2)

BINOR(aexp-1,aexp-2)

BINSUB(aexp-1,aexp-2)

BINXOR(aexp-1,aexp-2)

where:

aexpl-2 are arithmetic expressions,
variables, or constants.

The BINAND, BINOR, BINXOR functions perform logical operations bit by bit on 16-bit operands. The BINADD and BINSUB functions perform binary arithmetic operations on integer (16-bit) operands.

BINADD performs the BINARY ADDition of aexp-1 and aexp-2. A carry is ignored.

BINAND performs a BINARY AND logical operation on aexp-1 and aexp-2. It returns a 1 bit in a given position if both bits are equal to 1 and a 0 otherwise.

BINOR performs a BINARY OR logical operation on aexp-1 and aexp-2. It returns a 1 bit in a given position if either bit is equal to 1 and a 0 otherwise.

BINSUB performs the BINARY SUBtraction of aexp-1 minus aexp-2. An overflow is ignored.

BINXOR performs a BINARY EXclusive OR logical operation on aexp-1 and aexp-2. It returns a 1 bit in a given position if either bit is equal to 1 and the other bit is equal to 0. A 0 is returned otherwise.

Notes:

1. If necessary, aexp-1 and/or aexp-2 will be converted to 16 bit integers for the purpose of these functions.

16.1 Arithmetic Functions

2. Refer to section 6.4 for a discussion of Boolean operators.

16.1 Arithmetic Functions

function: Exponent

format: EXP(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The EXP function calculates the value of the
constant e (where $e = 2.71828\dots$) raised to the
aexpth power.

Example:

>>List

```
10 Num1=4.1
20 Num2=Exp(Num1)
30 Print Num2
40 End
```

>>Run

```
60.340287597344
***40 End***
```

function: Fractional Portion

format: FRA(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The FRA function returns the FRActional portion of
aexp.

Example:

>>List

```
10 Number=3.7
20 Print Fra(Number)
30 End
```

>>Run

0.7

30 End

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.1 Arithmetic Functions

function: Integer Portion

format: INT(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The INT function returns the largest INTEGER value
which is less than or equal to aexp.

Examples:

>>List

```
100 Boxes=5.7
200 Print Int(Boxes)
300 End
```

>>Run

```
5
***300 End***
>>Print Int(-5.7)
-6
```

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.1 Arithmetic Functions

function: Integer Random Number Generator

format: IRN(X)

where:

X is a dummy argument.

The IRN function generates an Integer Random Number between 0 and +32767.

Note:

1. To change the sequence of random numbers, the RANDOMIZE instruction should be included in the program.

Example:

>>List

```
10 Rem Demo of Integer Random Number Generator
20 For Section=1 to 10
30 Print Irn(6)
40 Next Section
50 End
```

>>Run

29284

25801

18835

4647

9295

18846

4924

10105

20210

7652

40 End

The program listed above will print out 10 integer random numbers. If this program is run a second time, it will generate the same 10 random numbers. A different set of random numbers will be generated only if a RANDOMIZE statement is included at the beginning of the program.

16.1 Arithmetic Functions

function: Logarithm

format: LOG(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The LOG function calculates the natural LOGarithm (i.e., log to the base e) of aexp.

Note:

1. The logarithm base 10 for X can be computed as follows:

$$\text{Log'base'10} = \text{Log}(X) / \text{Log}(10)$$

Example:

>>List

```
10 Input Number
20 Nat'log=Log(Number)
30 Print Nat'log
40 End
```

>>Run

? 3.2

1.1631508098056

40 End

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.1 Arithmetic Functions

function: Maximum Value

format: MAX(aexp-1,...,aexp-n)

where:

aexpl-n are arithmetic expressions,
variables, or constants.

The MAX function examines the list of arithmetic expressions (aexp-1 through aexp-n) and returns the value of the largest expression.

Example:

>>List

```
10 Num1=10
20 Num2=25
30 Max'val=Max(Num1,Num2)
40 Print Max'val
50 End
```

>>Run

25

50 End

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.1 Arithmetic Functions

function: Minimum Value

format: MIN(aexp-1,...,aexp-n)

where:

aexp1-n are arithmetic expressions,
variables, or constants.

The MIN function examines the list of arithmetic expressions (aexp-1 through aexp-n) and returns the value of the smallest expression.

Example:

>>List

```
10 Num1=5
20 Num2=10
30 Min'val=Min(Num1,Num2)
40 Print Min'val
50 End
```

>>Run

```
5
***50 End***
```

instruction: Randomize

format: [Ln] RANDOMIZE

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The RANDOMIZE instruction is used to reset the random number dummy variable used by the RND and IRN functions so that a different sequence of random numbers will be produced each time the RND and IRN functions are used.

Notes:

1. RANDOMIZE should be used only once within a program utilizing RND to ensure that a truly random sequence of numbers results.
2. RANDOMIZE should be used after generating every 1000 numbers using IRN. This will ensure a truly random sequence of numbers.

Example:

>>List

```
10 Print "This is a Random number ";
20 Randomize
30 Print Rnd(0)
40 End
```

>>Run

```
This is a Random number 0.7137712225
***40 End***
```

>>Run

```
This is a Random number 0.8171978025
***40 End***
```

This program will PRINT a different random number everytime it is RUN.

16.1 Arithmetic Functions

function: Random Number Generator

format: RND(X)

where:

X is a dummy argument.

The RND function generates a RaNDom number in the range $0 \leq \text{Rnd}(0) < 1$.

Examples:

>>List

```
10  Rem Demo of Random Number Generator
20  For Index=1 To 100
30  Print Rnd(2)
40  Next Index
50  End
```

The above example will PRINT 100 random numbers. If this program is RUN a second time, it will generate the same 100 random numbers. To generate a new set of random numbers each time the program is RUN, the above example can be rewritten as follows:

>>List

```
10  Rem Demo of Random Number Generator
15  Randomize
20  For Index=1 To 100
30  Print Rnd(2)
40  Next Index
50  End
```

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.1 Arithmetic Functions

function: Sign

format: SGN(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The SGN function returns a +1 if the value of the
expression aexp is greater than 0, a 0 if aexp
equals 0, and a -1 if aexp is less than 0.

Example:

>>List

```
10 Input Value1,Value2,Value3
20 Print Sgn(Value1)
30 Print Sgn(Value2)
40 Print Sgn(Value3)
50 End
```

>>Run

? -12,0,14

-1

0

1

50 End

16.1 Arithmetic Functions

function: Square Root

format: SQR(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The SQR function calculates the Square Root of the positive expression aexp.

Example:

```
>>Print Sqr(9)  
3
```

function: Arctangent

format: ATN(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The ATN function calculates the ArcTaNgent of
aexp.

Note:

1. Although Structured Basic does not include predefined Arcsin and Arccos functions, the user can calculate these using the ATN function as follows:

$$\begin{aligned}\text{Arcsin}(X) &= \text{Atn}(X/\text{Sqr}(-X*X+1)) \\ \text{Arccos}(X) &= -\text{Atn}(X/\text{Sqr}(-X*X+1))+2.*\text{Atn}(1.)\end{aligned}$$

Example:

```
>>Print Atn(.80)
0.67474094222353
```

In this example the ArcTaNgent of .80 is equal to .6747094222353 radians. If the DEGREE mode is selected, the arctangent of .80 is given in DEGrees:

```
>>Deg

>>Print Atn(.80)
38.659808254087
```

16.2 Trigonometric Functions

function: Cosine

format: COS(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The COS function calculates the COSine of the angle represented by aexp.

Notes:

1. See ATN for a description of how to calculate Arccos.
2. Unless DEG mode has been selected, it is assumed that the value of aexp is expressed in RADians.

Example:

>>List

```
10 Input Num1
20 Let Num2=Num1*2
30 Print Cos(Num2)
40 End
```

>>Run

? .60

0.36235775447529

40 End

In this example, the COSine of a 1.20 radian angle is .36235775447529.

function: Sine

format: SIN(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The SIN function calculates the SINE of an angle represented by aexp.

Notes:

1. See ATN for a description of how to calculate Arcsin.
2. It is assumed that the value of aexp is expressed in RADIANS unless the DEG mode has been specified.

Examples:

>>List

```
10 Input Number
20 Print Sin(Number*3)
30 End
```

>>Run

```
? .04
0.11971220728892
***30 End***
```

In this example, the SINE of a .12 radian angle is approximately .12.

In the following example, we first select the DEG mode and then request the program to PRINT the SINE of a 90 degree angle:

>>List

```
5 Deg
10 Print Sin(90)
20 End
```

>>Run

```
1
***20 End***
```

The SINE of a 90 degree angle is 1.0.

16.2 Trigonometric Functions

function: Tangent

format: TAN(aexp)

where:

aexp is an arithmetic expression,
variable or constant.

The TAN function calculates the TANGent of the angle represented by aexp.

Note:

1. It is assumed that the value of aexp is expressed in RADians unless the DEGree mode has been specified.

Example:

>>List

```
5   Deg
10  Print Tan(30)
20  End
```

>>Run

0.577350269189

20 End

16.3 Programmer Defined Functions

function: Programmer Defined Function

format: DEF FNS(avar-1,avar-2,...,avar-n)=aexp

where:

S is any legal arithmetic variable name.

avar1-n are arithmetic variables.

aexp is an arithmetic expression, variable, or constant.

The DEF FuNction permits the programmer to define functions in addition to the pre-defined functions included in Basic.

Notes:

1. The definition of a function must be a statement which is encountered during the execution of a program in order for Basic to retain the definition. After the function has been defined, it may be used in a command line.
2. Any of the variables used in the definition of a function (avar-1 through avar-n) are unique to the function. Their use in the definition does not conflict with a variable of the same name appearing elsewhere in the program.

If a variable appears on the right side of the definition (as a part of aexp) without appearing on the left, its value may be accessed and changed outside the function definition and its value is maintained throughout a function call. These variables are not unique to the function.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.3 Programmer Defined Functions

Examples:

>>List

```
10 Length=20
20 Width=10
30 Height=5
40 Def Fnvolume(A,B,C)=A*B*C
50 Print Fnvolume(Length,Width,Height)
60 End
```

>>Run

```
1000
***60 End***
```

In the above example, variables A, B, and C are unique to the definition of the function. Variables Height, Length, and Width are regular program variables whose values may be established at any point in the program (Partition).

The following example should help to clear up any confusion about variables which are unique to a function definition.

>>List

```
100 Def Fnmult(P'meter) = P'meter * Factor
200 P'meter = 77
300 Factor = 5
400 Actual'value = 10
500 Print Fnmult(Actual'value)
600 Print P'meter
700 End
```

>>Run

```
50
77
***700 End***
```

In the above example, statement 100 defines the function Fnmult in terms of the formal parameter (function definition variable) P'meter and the program variable Factor. In statement 100 P'meter is called a function definition variable because it appears to the left of the equal sign in the function definition. As such, P'meter can not be accessed by the user. The program variable (P'meter) defined by statement 200 is a different variable than the aforementioned function definition variable, and can be accessed by the

16.3 Programmer Defined Functions

user. According to the above definitions, Factor, appearing in statement 100, is a program variable. In statements 300, 400, and 500 program variables P'meter, Factor, and Actual'value are assigned values.

In statement 500, the value of the function Fnmult is computed using the value of Actual'value to replace P'meter in the definition of the function. Being a program variable, Factor maintains its value in the calculation of the value of the function. Once the value of the function has been calculated, its value is printed.

Statement 600 prints the value of the program variable P'meter to demonstrate that the program variable P'meter can be accessed by the user, that it maintains its value through the function call, and that it is a different variable than the function definition variable P'meter.

16.4 String Functions

function: ASCII Value of a Character

format: ASC(svar)

where:

 svar is a string variable or literal.

The ASC function returns the ASCII decimal value of the first character of string.

Note:

1. Refer to the Appendix for a table of ASCII characters and their values.

Example:

>>List

```
10 Input Text$
20 Print Asc(Text$)
30 End
```

>>Run

? A

65

30 End

In this example, the ASCII decimal value for the first character of the string variable Text\$ (character A) is 65.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.4 String Functions

function: Character

format: CHR\$(aexp)

where:

aexp is an arithmetic expression,
variable, or constant.

The CHR\$ function returns the single CHaRacter
which is represented in ASCII by aexp.

Notes:

1. If aexp is outside of the range $0 \leq aexp \leq 255$ an error will be generated.
2. This function allows the user to draw graphs or figures with special characters. The function may also be used to initiate special functions such as cursor positioning, generating line or form feeds, or causing a bell to sound on the terminal.
3. This function may be used to output non-printing and special purpose characters such as control or underline characters.
4. CHR\$ may be used any place a string is required.

Example:

>>List

```
10 Input Value
20 Print Chr$(Value)
30 End
```

>>Run

```
? 42
*
***30 End***
```

The ASCII decimal value 42 is equivalent to the character *. Thus, in the above example, the instruction PRINT CHR\$(42) instructs the computer to output the character * on the terminal.

16.4 String Functions

instruction: Expand String

format: [Ln] EXPAND svar, exp-2

[Ln] EXPAND svar(exp-1), exp-2

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable.

exp-1 is an optional arithmetic expression, variable, or constant. The default value is $\emptyset$.

exp-2 is an arithmetic expression, variable, or constant.

The EXPAND instruction inserts null characters into a string variable. The number of nulls to be inserted is specified by exp-2. The nulls are inserted before the character specified by exp-1, or, if exp-1 is omitted, before the first character of the string.

Notes:

1. Remember that the first character in a string is located at index position $\emptyset$ (svar($\emptyset$)).
2. This instruction is very useful for inserting characters into the middle of a string.

Example:

>>List

```
100 Word$ = "ABEF"
110 Expand Word$(2),2
120 Word$(2,3) = "CD"
130 Print Word$
```

>>Run
ABCDEF

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.4 String Functions

function: ASCII Hex Representation

format: HEX\$(avar)

where:

avar is an arithmetic variable or constant.

The HEX\$ function will convert avar into its 4 byte Hexadecimal ASCII (string) representation.

Example:

>>List

```
100 Number=32
200 String$=Hex$(Number)
300 Print String$
400 End
```

>>Run

0020

400 End

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
16.4 String Functions

function: Length of String

format: LEN(svar)

where:

 svar is a string variable or literal.

The LEN function returns an integer value which is equal to the number of characters in any string variable svar. In other words, the LEN function gives the LENgth of a string.

Note:

1. Both characters and spaces are counted as part of the length. Trailing null characters are not counted as part of the string length.

Example:

>>List

```
10 Dim String1$(20),String2$(30)
20 Input String1$,String2$
30 @"Length of String1 is ";Len(String1$)
40 @"Length of String2 is ";Len(String2$)
50 End
```

>>Run

```
? Example
?? Length Command
Length of String1 is 7
Length of String2 is 14
***50 End***
```

function: Position of Substring

format: POS(svar-1,svar-2,aexp)

where:

svar-1 is a string variable or literal

svar-2 is a substring variable or
literal

aexp is an arithmetic expression,
variable, or constant

The POS function is used to locate the POSition of a substring (svar-2) within a string (svar-1). The position within the string svar-1 at which the search is to begin is specified by the arithmetic expression aexp. This function returns a value equal to the POSition index of the first character of the substring within the string.

Notes:

1. A -1 is returned if the substring is not found.
2. Remember that the first element of a string is numbered 0.

Example:

>>List

```
10 Dim String$(50)
20 String$="This is a substring search"
30 String'is=Pos(String$,"is",4)
40 String'r=Pos(String$,"r",20)
50 Print String'is
60 Print String'r
70 End
```

>>Run

5

23

70 End

In this example, the computer is first instructed to search for "is" starting from the fourth

16.4 String Functions

character in string\$ and second to search for "r" starting from the twentieth character in string\$. Starting from position 4, the first character in substring "is" is located in position 5. Starting from position 20, the first character "r" is located in position 23. Consequently, the computer returns a value of 5 for String'is and 23 for String'r.

An application of this function is the conversion of numbers from hexadecimal to decimal. The following program demonstrates the principle using a one digit number:

>>List

```
1000 Dim Hex'number$(0), Answer$(0)
1100 *Start : Input "One digit hex number: ",Hex'number
1200 Decimal'number = Pos ("0123456789ABCDEF", Hex'number, 0)
1300 IF Decimal'number = -1 Then Goto Start
1400 Print "The decimal equivalent is: ";Decimal'number
1500 Input "Another one (Y/N)? ",Answer$
1600 If Answer$ = "y" then goto Start
1700 If Answer$ = "Y" then goto Start
1800 End
```

function: String Equivalent

format: STR\$(aexp)

where:

aexp is an arithmetic expression,
variable, or constant.

The STR\$ function converts an expression (aexp) to a STRING which is the ASCII representation of the expression.

Note:

1. Valid input to this function includes the decimal point (.) and a leading plus (+) or minus (-) sign.
2. STR\$ may be used any place a string is required.

Example:

>>List

```
10 Input Number
20 String$=Str$(Number)
30 Print String$
40 End
```

>>Run

? 8.45

8.45

40 End

16.4 String Functions

function: Value of String

format: VAL(svar)

where:

svar is a string variable or literal.

The VAL function converts a string (svar) into a numeric variable which may be used in an arithmetic expression.

Notes:

1. If the argument string for VAL consists of both numeric and non-numeric information, the following conventions hold:
 - a. If the first character is non-numeric, VAL will return a zero value (this can be used to decode a user's input). The first character is considered to be numeric if it is the leading percent sign (%) of a hexadecimal constant.
 - b. If the first character or characters are numeric they will be converted without consideration of the portion of the string including and following the first non-numeric character.

Example:

>>List

```
10 String$="26.6321"
20 Increment=1
30 Value=Increment+Val(String$)
40 Print Value
50 End
```

>>Run

```
27.6321
***50 End***
```

function: Value of String
 With Error Checking

format: VALC(svar)

where:

 svar is a string variable or literal.

The VALC function returns a value which can be assigned to a numeric variable. This function provides a user trappable error when its argument is not properly structured.

Notes:

1. The error conditions that may be set are as follows:
 - A. The actual parameter of a VALC function call has evaluated to a numeric quantity whose accuracy is outside the range of Long Floating Point numbers.
 - B. The actual parameter of a VALC function call has evaluated to a numeric quantity whose value is outside the range of Long Floating Point numbers.
 - C. The actual parameter of a VALC function call was not structured according to the proper syntax for numeric constants.

The characteristic aspects of this syntactic specification are as follows:

1. The comma is not allowed to exist in numbers.
 2. If the string contains a decimal point there must be digits on both sides of it.
 3. Both integers and floating point numbers may contain exponents.
 4. Multiple positive or negative signs are illegal.
2. If the process of converting the string is successful, the function will return a numeric

16.4 String Functions

value which is assignable to a numeric data type. If the string is properly structured, but evaluates to a value outside of the accuracy (condition A) or range (condition B) of the Long Floating Point numbers, then a run-time trappable error will result. The function may also be aborted for a syntax error (condition C). If it is aborted for any reason, the returned value is undefined.

If the value is to be assigned to a variable of a less precise data type, the value will be rounded as per Basic's implicit type conversion.

17. System and File Status

command: Disk Drive

format: DSK
DSK "X"

where:

X is a disk drive specifier (1, 2, 3, 4, A, B, C, or D), the at sign, or zero (@ or Ø).

The DSK instruction displays or alters the current system (CDOS) default disk drive.

Notes:

1. When used without a disk drive specifier, DSK displays the current disk drive identifier.
2. While a program is RUNning, DSK will not display anything. Use IOSTAT(Ø,Ø) to determine the current drive while in the RUN mode.
3. When used with a disk drive specifier, DSK changes the current system default disk drive. New disks are not logged in by changing the default disk drive. See the following note for the method of logging in a new disk from Basic. The following is a list of equivalent disk drive specifiers:

A: or 1
B: or 2
C: or 3
D: or 4

The above specifiers may be enclosed in quotation marks (string literal) or assigned to a string variable.

4. When used with the at sign or zero (@ or Ø), DSK logs in a new disk (after it has been inserted in any drive) and causes drive A to become the current system default disk drive. The effect of this is similar to typing CTRL-C while in CDOS. The following instructions will preserve the current drive while performing a logical CTRL-C:

I=Iostat(Ø,Ø) : Dsk "Ø" : Dsk Chr\$(I)

17. System and File Status

instruction: Enable Echo

format: [Ln] ECHO

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The ECHO instruction is used to re-enable the display of certain information at the console terminal after the display has been disabled by the NOECHO instruction.

17. System and File Status

instruction: Disable Echo

format: [Ln] NOECHO

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The NOECHO instruction is used to disable the display of user entered responses to the INPUT instruction.

Notes:

1. This instruction is useful when a secret code or password is to be INPUT. The code will not be displayed on the screen, making theft of the code almost impossible.
2. The NOECHO mode is not reset by the RUN or SCRatch instructions. It is reset by the ECHO instruction.

Example:

>>List

```
100 Noecho
200 Input Secret'number
300 Print Secret'number
400 End
```

>>Run

```
? (user entered INPUT is not displayed)
517 (the value of Secret'number is PRINTed)
***400 End***
```

instruction: Enable Escape

format: [Ln] ESC

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The ESC instruction is used to re-enable ESCape key operation after it has been disabled by the NOESC instruction.

17. System and File Status

instruction: Disable Escape

format: [Ln] NOESC

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

The NOESCape instruction disables the console terminal escape key operation.

Notes:

1. Most terminal keyboards include a key labeled ESC. Basic recognizes the ESCape character as a signal to abort program execution and return to the command mode. The NOESC instruction is used to prevent program interruption when the ESCape key is pressed.
2. ESCape results from the use of the CTRL-Z, ESCape, or CTRL-[key.
3. ESCape key operation is reset by the ESCape instruction.
4. Most programs will execute significantly faster when the operation of the ESCape key has been disabled by the NOESC instruction.

function: Free Space

format: FRE(X)

where:

X is a dummy argument,

The FRE function gives the number of bytes of memory in the User Area which are currently FREe or unused.

Notes:

1. Certain statements do not occupy their full space until after they have been executed (e.g., DIM).
2. Because Basic allocates space for its internal tables in segments, the FRE function is only an approximation of the actual number of bytes available to the user.
3. Space recovered from DELETED lines is available for new program lines, not for arrays or variables. Thus, the FRE function will not necessarily reflect available program space.

If statement lines have been DELETED from the User Area, it will be necessary to perform the following steps in order for the FRE function to reflect the change:

- a) LIST the program (do not SAVE it) to a temporary disk file.
- b) SCRatch the User Area.
- c) ENTER the temporary disk file.
- d) Give the command PRINT FRE(X).

17. System and File Status

function: I/O Status

format: IOSTAT(aexp-1, aexp-2)

where:

aexp-1 is a file (channel) number.

aexp-2 is the status parameter.

The IOSTAT function returns the current status of an OPENed Basic file.

<u>Status Parameter</u>	<u>Returned Value</u>	<u>Meaning</u>
0	0	File is ok.
	1	End of File.
	2	Attempt to read an unwritten segment during a random access read.
1	X	X is the current sector number. (This is not the record number.)
2	Y	Y is the current byte number within sector X (above).

Notes:

1. Devices other than disk drives may or may not return status values.
2. IOSTAT(0,0) will return the current disk number. Disk A=1, B=2, C=3, and D=4.
3. If Rec is the record size and File is the file number, the expression:

$\text{INT}((128.0 * \text{Iostat}(\text{File},1) + \text{Iostat}(\text{File},2))/\text{Rec})$

will give the current record number. A similar expression can derive the current byte within the record.

17. System and File Status

instruction: On Error Transfer Control

format: [Ln] ON ERROR STOP

[Ln] ON ERROR GOTO n

[Ln] ON ERROR GOSUB n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is the line number or line name of the statement to which control is transferred.

The ON ERROR instruction causes program control to be transferred as specified (STOP, GOTO, or GOSUB) when a non-fatal error occurs during program execution.

Notes:

1. A non-fatal error in Basic is any error listed in the error table (see the Appendix) with a number of 128 or greater. Errors numbered 127 and below are defined as fatal errors and cannot be trapped with an ON ERROR statement.
2. If ON ERROR is written at the beginning of a program, the instruction specified with ON ERROR will be executed each time a program error occurs. If placed elsewhere in the program, the instruction will be executed only for errors which occur during the execution of statements following the execution of the ON ERROR statement.
3. ON ERROR is reset by the RUN, SCRatch, and ON ERROR STOP instructions.
4. Using RETRY to exit from a subroutine will cause the instruction which caused the error to be executed again. Refer to the RETRY instruction.

17. System and File Status

Example:

```
60    Input Num1,Num2
80    Print Num1*Num2
100   On Error Goto 300
120   Input Num3,Num4
140   Print Num3/Num4
160   Goto 60
300   Print "A non-fatal error has occurred"
320   Goto 120
```

In this example, any error which occurs before line 100 has been executed for the first time will be dealt with by the standard system error handling procedure. Any trappable error which occurs after line 100 has been executed will cause program execution to continue with statement line 300.

17. System and File Status

instruction: On Escape Transfer Control

format: [Ln] ON ESC STOP

[Ln] ON ESC GOTO n

[Ln] ON ESC GOSUB n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

n is the line number or line name of the statement to which control is transferred.

The ON ESC instruction causes program control to be transferred as specified (STOP, GOTO, or GOSUB) when the ESCape key is depressed during program execution.

Notes:

1. If ON ESC is written at the beginning of a program, the instruction specified with ON ESC will be executed each time the ESCape key is depressed. If placed elsewhere in the program, the statement will be executed only when the ESCape key is depressed following the execution of the ON ESC statement.
2. ON ESC is reset by the RUN, SCRatch, and ON ESC STOP instructions.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
17. System and File Status

Example:

>>List

```
10 *Begin : Input Cond1,Cond2,Cond3
20   If Cond1 = -1 Then 70
30   Let Answer=Cond1+Cond2+Cond3
40   On Esc Goto Begin
50   Print Answer
60 *Wait : Goto Wait
70   End
```

>>Run

? 10,15,20

45

(ESCAPE must be pressed here to
get out of the Wait loop.)

? 70,10,5

85

? -1,0,0

70 End

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
17. System and File Status

instruction: Set System Parameter

format: SET aexp-1, aexp-2

where:

aexpl-2 are arithmetic expressions,
variables, or constants.

aexp-1 is the system parameter
number.

aexp-2 is the value to be
assigned to the
parameter.

The SET instruction is used to change the value of
a given system parameter.

Notes:

1. Refer to the SYS instruction for a list of all
user accessible system parameters.

Refer to the Appendix, Areas of User Interest,
if it is necessary to change the default
values of any of the system parameters.

2. System parameter number 0 is the width of a
page in characters and may assume any value
between 0 and 32767. The default value for
this parameter is 80 characters which
corresponds to a standard 8 1/2 inch page.

A special use of the SET command is:

Set 0, -1

which inhibits the automatic carriage RETURN-
LINE FEED at the end of a line when page width
is reached. Refer to the discussion of the
PRINT instruction as it pertains to file
output.

Using SET 0,-1 to disable page width checking
is especially useful for graphics output to
devices such as the Cromemco 3355 printer.

3. System parameter 1 is the tab field width and
may assume any value between 1 and 32767.

17. System and File Status

This corresponds to the width of the field which is output by a comma (,) in a PRINT instruction. If the tab field width is SET to a 0 or -1, no tabbing takes place (i.e., commas are treated as semicolons). The default value for this parameter is 20 characters.

4. System parameter 5 may be used to facilitate timed input. To start the timer, the SET 5 instruction is given, with aexp-2 equal to approximately ten (10) times the number of seconds of delay desired. Halve aexp-2 if using a 2MHz clock. For example:

Set 5,50

will allow about 5 seconds for a complete user response.

When the next INPUT instruction is encountered, Basic will issue an Error 210 -- INPUT Timeout if the user does not respond with a complete INPUT within the allotted time.

Once SET, the time used to respond to all subsequent INPUT instructions is added together until it exceeds the specified value, and then an error message is generated. It is therefore necessary to give the SET instruction prior to each timed INPUT. When timed INPUT is no longer desired, it may be de-activated by coding:

Set 5,0

The ON ERROR statement can be used to trap the timeout error. The programmer may find out how much time was used by coding SYS(5) to find the time remaining.

17. System and File Status

function: System Parameter

format: SYS(aexp)

where:

aexp is an arithmetic expression,
variable, or constant.

The SYS function provides system information based on the value of the argument aexp.

Notes:

1. See the SET instruction for methods of changing the values of these system parameters.

Refer to the Appendix, Areas of User Interest, if it is necessary to change the default values of any of the system parameters.

Example:

```

100 On Error Goto Error'routine
.
.
140 Open \1\ "FILE.DAT"
.
.
699 *Error'routine
700 If Sys(3)=128 Then Create"FILE.DAT":Goto 140
.
.

```

In this example, if the file FILE.DAT did not originally exist on the disk, the error routine at line 700 would CREATE the file. Lines 710, 720 etc. could be written to test for other values of SYS(3), and take appropriate action(s).

The table on the following page lists all of the user accessible system parameters.

The first column represents the Value (aexp) of the System Parameter. SET indicates if it is valid to SET this parameter (Yes) or if it may just be examined by the SYS function (No). The Range and Default are those of the Parameter while the last column is a brief description of the Parameter.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
 17. System and File Status

<u>Value</u>	<u>SET</u>	<u>Range</u>	<u>Default</u>	<u>System Parameter</u>
0	Yes	<u>>0</u>	80	Page width
1	Yes	<u>>0</u>	20	Tab field width
2	No	0-255	-	character last printed
3	No	0-255	-	Last run time error
4	Yes	0-Sys(0)	-	Current print column
5	Yes	any	0(off)	INPUT timeout
6	Yes	0,1	0	Upper case LISTing
7	Yes	0,7	0	LIST constants with spaces
8	Yes	-2 thru 5	0	Indentation control
9	Yes	0,1	0	Ignore quotation marks on INPUT
10	No	0-7	-	Current Partition number
11	Yes	1-15	4	Basic-KSAM Pages per Data Block
12	Yes	1-15	4	Basic-KSAM Pages per Key Block
13	reserved			
14	reserved			
15	Yes	any	-	Reserved for user inter-program communication

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
18. Machine Level Instructions

function: Address Of A Variable

format: ADR(var)

where:

var is a string or arithmetic
variable.

The ADR function will return the memory storage
location starting address of a variable.

Note:

1. Addresses above 32,767 will be represented as
negative numbers. They are unsigned 2 byte
integers.
2. The ADR function allows the use of subscripted
variables.

18. Machine Level Instructions

instruction: Input From I/O Port

format: INP(m)

where:

m is an I/O port.

The INP instruction is used to read the contents of INPUT port (m).

Note:

1. The value of m must be in the range $0 \leq m < 255$.

Example:

```

10  Noesc : Integer Character
20  *Poll : If Binand(Inp(0),%0040%) = 0 Then Goto Poll
30  Character = Inp(1)
40  Character = Binand(Character,%007F%)
60  If Character = %001B% Then Print : Goto Escape
70  Print Chr$(Character);
80  If Character = 13 Then Print
90  Goto Poll
100 *Escape : Esc
110 End

```

When a character is typed at the console, this program will echo it back to the console CRT. A RETURN will be echoed as a RETURN followed by a LINE FEED, and an ESCape will terminate the program.

The NO ESCape instruction at line 10 is necessary to disable Basic's continuous polling of input port 0. While a program is being executed, Basic ignores most characters typed at the console. Basic continually polls the UART status port 0 and if a character is ready, reads it from port 1. If this character is not an ESCape character, it is ignored by Basic. If it is an ESCape character, program execution is terminated.

Upon reading the UART status, the status bit is reset. It is necessary to disable Basic's polling of the status port so that this user program will be able to determine if a character has been typed. If this were not done, Basic would, by reading the

18. Machine Level Instructions

status port, reset the character ready bit, thereby preventing the user program from determining that a character had been typed.

Line 60 replaces the ESCape key function by testing the character typed by the user. If it is an ESCape, program execution is terminated.

Statement 20 is a loop which continually polls the UART status ready port to determine if a character has been typed. The Receiver Data Available flag is bit 6 (refer to the 4FDC manual). The contents of input port 0 is ANDed with 40 hex to determine if this bit is a one or a zero. If it is a zero, no data is available, and control remains on line 20, polling the input port again.

If the bit has been set (=1) it indicates that data is available, and statement 30 reads the data from input port 1 and assigns this data to the variable Character. Line 40 strips the parity bit off of the input data (the ASCII character set requires only the seven bits, 0 through 6) and as stated before, line 60 terminates program execution upon determining that an ESCape character (1B hex) has been INPut.

The rest of the program prints the ASCII character whose value was INPut, PRINTs a LINE FEED if a RETURN was INPut, and transfers control to line 20.

When an ESCape is detected, control is transferred to line 100 which re-enables the operation of the ESCape key and terminates program execution.

instruction: Output To I/O Port

format: OUT m,b

where:

m is an I/O port

b is a byte value

The OUT instruction is used to OUTput data byte (b) to OUTput port(m).

Notes:

1. The value of m must be in the range $0 \leq m \leq 255$.
2. The value of b must be in the range $0 \leq b \leq 255$.

Examples:

>>Out 1,75

will display the character K (ASCII 75) on the console terminal (output port 1).

>>Out 52,0

will turn off the disk drive motors while

>>Out 52,255

will turn them back on.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
18. Machine Level Instructions

function: Peek At Memory

format: PEEK(m)

where:

m is a memory location.

The PEEK function returns the contents of memory location (m).

Note:

1. If m is decimal, it should be in the range: $0 \leq m \leq 32767$. Hexadecimal numbers may be used to access locations in the range: $0 \leq h \leq \%FFFF\%$. Negative decimal numbers map into their hexadecimal equivalents:

Peek(-2)

is the same as

Peek(%FFFE%)

Example:

Print Peek(5)

will PRINT the contents of memory location 5.

function: Poke Into Memory

format: POKE m,b

where:

m is a memory location.

b is a byte value.

The POKE instruction puts the byte value b into memory location m.

Notes:

1. If m is decimal, it should be in the range: $0 \leq m < 32767$. Hexadecimal numbers may be used to access locations in the range: $0 \leq h < \%FFFF\%$. Negative decimal numbers map into their hexadecimal equivalents:

Poke -2,5

is the same as

Poke %FFFE%,5

2. The value of b must be in the range $0 \leq b < 255$.

Example:

Poke 1000,255

will place the value 255 into memory location 1000.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
18. Machine Level Instructions

function: Call a User Program

format: USR(m,P1,...,Pn)

where:

m is the address of the assembly language routine.

P1-n are parameters which are converted to 16 bit integers.

The USR function makes it possible to call an assembly language subroutine from a Basic program.

Notes:

1. The USR function always requires the user to specify one parameter in addition to the address, even if it is a dummy parameter. For example, USR(0,1) is correct, while USR(0) will result in a syntax error.
2. If m is decimal, it must be in the range: $0 \leq m \leq 32767$. Hexadecimal numbers may be used to access locations in the range: $0 \leq h \leq \text{\%FFFF\%}$.
3. USR is a function and must take the form of a function when it is used:

>>100 Alpha = Usr(X,Y)

4. When the user routine gains control (at the address specified in the USR function call), the following conventions apply:
 - a) Register A contains the number (n) of parameters in the function call.
 - b) Register pair HL contains the return address to Basic. The user routine may re-enter Basic by coding the following assembly language instruction:

JP (HL)

- c) The parameters were placed in order (i.e., P1 was Pushed first, followed by P2, P3,..., Pn) on the CPU stack and may be recovered via Pop instructions.

- d) If and only if n parameters (n is the contents of register A, as above) are Popped off the stack, and the stack pointer is not otherwise changed, Basic may be re-entered via the following instruction:

RET

- e) The routine may return a 16-bit value to be assigned to the function by placing the value in DE register pair before re-entering Basic.
- f) Aside from the restrictions noted above, all registers may be used in any way by the user routine.

Example:

The following routine, when POKEd into position by the Basic program starting at line 100, will allow the Basic user to perform CDOS system calls. The USR function must be used to access this routine. The first parameter of the USR function is the starting address of the assembly language routine (in this case, 103 Hex). The second and third parameters are the values which are to be loaded into the DE and BC register pairs before the system call. Upon returning from the routine, the entire USR function takes on the value of the A register. This value may be accessed by assigning the USR function to a variable. Notice that the C portion of the BC register pair always contains the CDOS system call number.

18. Machine Level Instructions

```

89   Rem Data statements 100 & 110 are the machine
90   Rem code equivalent of the following assembly
91   Rem language routine:
92   Rem
93   Rem pop de
94   Rem pop bc
95   Rem call 5
96   Rem ld d,0
97   Rem ld 'e,a
98   Rem ret
99   Rem
100  Data %00D1%,%00C1%,%00CD%,%0005%,%0000%
110  Data %0016%,%0000%,%005F%,%00C9%
120  Integer Byte,Index,Dummy,Row
130    For Index = %0103% To %010B%
140      Read Byte
150      Poke Index,Byte
160      Next Index
170  Rem
1000  Rem Following Basic program demonstrates
1100  Rem the CDOS system call which positions the
1200  Rem CRT cursor.
1300  Rem
1400  Set 0,-1 : Rem disable page width checking.
1500  Rem clear screen and home cursor.
1600  Dummy = Usr(%0103%,0,142)
1700    For Row = 1 To 24
1800      Gosub 4400
1900    Next Row
2000  Rem
2100  Rem Clear and home cursor.
2200  Dummy = Usr(%0103%,0,142)
2300    For Row=24 To 1 Step -1
2400      Gosub 4400
2500    Next Row
2600  Goto 1600
2700  Rem
2800  Rem
4000  Rem subroutine to compute diagonal column
4100  Rem given a row number, position cursor
4200  Rem at the given location, and print the
4300  Rem index number there.
4400  Column = Row*3 : Rem compute column number
4500  Rem Move column number into high byte
4600  Rem and row number into low byte
4700  Col'and'row = 256*Column+Row
4800  Dummy = Usr(%0103%,Col'and'row,142)
4900  Print Row;
5000  Return
5100  End

```

function: Type Of Variable

format: TYPE(aexp)

where:

aexp is an arithmetic expression or variable.

The TYPE function will return a value indicating the TYPE of exp.

Note:

The following values are returned by TYPE:

<u>Value</u>	<u>TYPE of Expression</u>
1	Integer
2	Short Floating Point
4	Long Floating Point

18. Machine Level Instructions

functions: Basic-KSAM Numeric Sorting Conversions

formats: IKEY\$(aexp)

FKEY\$(aexp)

KEYI(svar)

KEYF(svar)

where:

aexp is an arithmetic expression,
variable, or constant.

svar is a string variable or string
literal.

These four functions allow the user to store numeric information in character variables so that numeric fields of a data file may also be key fields. The conversion is such that the ordering of the records with regard to these key values will be the ordering of the key fields, interpreted as numeric quantities.

Notes:

1. The Ikey\$ function will take an integer variable argument and return two character positions of a string variable. When these two character positions are used as a key field Basic-KSAM will put them into the file in the proper order of the integer argument. Thus, integers may be used as keys, and will be sorted according to the integer value.
2. The Fkey\$ function performs the same operation as the Ikey\$ function, except it takes a Long Floating Point variable as an argument and produces eight characters of a string variable as a result. These eight characters also cause the record to be added to the Basic-KSAM file in the order of the numeric quantity of the argument of the function. Short Floating Point numbers may be utilized by ignoring the last four bytes returned.
3. The Keyi function performs the opposite operation of the Ikey\$ function. When a record is read into a string variable which

18. Machine Level Instructions

contains bytes that are really the converted form of an integer, this function will translate them into the integer representation and assign the value to an integer variable. Thus, the argument for this function is two bytes of a string variable and the result is of type integer.

4. The Keyf function performs the reverse of the Fkey\$ function, and thus will take from four to eight characters of a string variable as an argument and return a Basic Long Floating Point number which may be assigned to a Long or Short Floating Point variable.

19. Scope of Variables

statement: Common Storage Area Method I

format: Ln COMMON

where:

Ln is a line number.

This version of the COMMON statement may be used when chaining programs (using RUN statements in the program). Refer also to the BEGINCOMMON and ENDCOMMON instructions in the description of Common Storage Area Method II.

This statement reserves a Common Storage Area for variable storage. The size and contents of this area are determined by the string variables and array variables which have been explicitly DIMensioned (used in a DIM, INTEGER, SHORT, or LONG instruction) since the last RUN instruction or since Basic was loaded. The space is reserved in a byte-by-byte fashion without regard to variable type or length.

A subsequent RUN instruction will initialize all variables except those in the Common Storage Area. The reserved area will be assigned on a byte-by-byte basis to variables as they are DIMensioned in the new program (the program called by the RUN instruction).

The Common Storage Area will cease to be reserved if two RUN instructions are issued without an intervening COMMON statement.

Note:

1. It is up to the user to ensure that DIMensioned variable types match between programs. For example, if program A is as follows:

```

200 Dim Name$(25), Interest'rate(10,2)
300 Integer Time'period(100)
400 Common
500 Run "B"

```

then program B, if it has explicitly DIMensioned variables, must DIMension:

- a. 26 bytes of string (ASCII) variables

19. Scope of Variables

- b. 264 bytes of Long Floating Point variables (8 bytes each)
- c. 202 bytes of Integer variables (2 bytes each)

in that order. The names and lengths of the strings and arrays do not matter, as long as the types match. Program B could use either of the following sets of statements:

- a. 10 Dim Title\$(9), Reference\$(15), Table(10,2)
20 Integer Time'prior(25), Time'current(24)
30 Integer Time'future(49)
40 Common
- b. 10 Dim Last'name\$(1), First'name\$(1), Middle'name\$(1)
20 Dim Initials\$(19), Value(10,2)
30 Integer Buffer(100)
40 Common

Notice that the 27th through 290th bytes in the reserved area must be DIMensioned similarly in each program if the same indices are to be used in each program. This is true for all two and three DIMensional arrays.

19. Scope of Variables

statement: Common Storage Area Method II

format: Ln BEGINCOMMON

Ln ENDCOMMON

where:

Ln is a line number.

This version of the Common statement may be used to establish a Common Storage Area when using Procedures.

A Common Storage Area is provided so that the programmer may declare certain strings and arrays as global throughout a series of calling and called Procedures.

A global string or matrix variable is one which may be referenced from a Partition other than the one in which it was originally declared. The value of a global variable may be altered both by the calling as well as the called Procedure regardless of whether the Procedures reside in the same or different Partitions.

Variables which are not used as receiving parameters in a procedure are always global within one Partition.

Defining and Accessing the Common Storage Area:

Only matrices (1, 2, or 3 DIMENSIONAL) and string variables can be placed in the Common Storage Area.

The amount of Common Storage Area which is established by the main program determines the maximum amount of Common Storage Area for all called Procedures.

When the RUN command is given, before any Basic instruction is executed, there is an implicit BEGINCOMMON instruction.

If the user does not wish to have a Common Storage Area, an ENDCOMMON instruction should be coded in the main program before any definition of or references to subscripted variables. This will cause Basic to reserve a Common Storage Area of length zero (i.e., no Common Storage Area).

19. Scope of Variables

If the user wishes to have a Common Storage Area for string variables and arrays, this area may be reserved by DIMensioning the variables (or by referencing the subscripted variables which implicitly DIMensions them) and then coding the ENDCOMMON instruction in the main program (the program residing in Partition zero).

Note that ENDCOMMON is implicit in a call to a Procedure or a return from a Procedure (ENDPROC or ERRPROC).

Once ENDCOMMON has been executed (either explicitly or implicitly) the Common Storage Area has been reserved. Loading another module or transferring control to another Partition will not alter this area.

It is not possible to expand the Common Storage Area to a size greater than was originally defined in the main program. The size of the Common Storage Area remains fixed until another RUN instruction is executed.

In order to access the Common Storage Area from a called Procedure in another Partition, it is necessary to code the BEGINCOMMON and ENDCOMMON instructions in the called Procedure. In between these two instructions, any reference to subscripted variables will cause the elements of the array or string to reference the Common Storage Area.

Storage is allocated from the bottom of the Common Storage Area, sequentially, as subscripted variables are referenced. Note that string variables occupy one byte for each element (character), while elements of an Integer, Short or Long Floating Point array will occupy 2, 4, and 8 bytes respectively. When assigning subscripted variables to the Common Storage Area, it is the programmer's responsibility to ensure that each type of variable is assigned to a sequence of bytes which were originally that same type of variable.

Remember that all arrays start with element zero (i.e., an array which is DIMensioned as 20 contains 21 elements numbered 0 through 20).

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
19. Scope of Variables

Example:

```
>>Run
This is a message
5          10          15
***240 End***
>>List

110  Rem MAIN Program
120  Rem
130  Rem
140  Rem There is an implicit BEGINCOMMON instruction
150  Rem before the first program instruction.
160  Lib"comexamp"
170  Dim Strg$(19),Var(2)
175  Endcommon
180  Strg$="This is a message"
190  Var(0)=5 : Var(1)=10 : Var(2)=15
210  Rem The following array is NOT in COMMON.
220  Dim Address(100)
230  Call .Called'procedure
240  End
```

>>Use 7

>>List

```
100 Procedure .Called'procedure
110  Rem
120  Rem COMMON is not accessed until the BEGINCOMMON
130  Rem instruction is given.
140  Begincommon
150  Dim Message$(19),Table(2)
160  Endcommon
170  Print Message$
180  Print Table(0),Table(1),Table(2)
190  Endproc
200  End
```

instruction: Define Local Variable

format: [Ln] LOCAL var-1, Mat var-2,...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

var1-n are scalar, matrix, or string variables. Matrix variables must be preceded by the word MAT.

The LOCAL instruction defines a new variable (or string or explicitly DIMensioned matrix) with the same name as a previously defined variable. The previous value of the variable is saved for later use. This process may be repeated as many times as necessary (or as storage space permits). The previous value of a variable will be recalled when a control structure, which was pending before the LOCAL instruction was given, is completed.

Notes:

1. The control structures which will cause the previous value of a variable to be restored are ENDDO, ENDWHILE, NEXT, ENDPROC, ERRPROC, EXITPROC, RETURN, and UNTIL.
2. DO and ENDDO may be used as a do-nothing control structure to cause variables to be popped as desired.
3. When a variable is defined as LOCAL, the contents of the variable are zeroed (numeric) or set to null characters (string).
4. This explanation may help clarify the LOCAL instruction to those users who are familiar with LIFO stacks. The LOCAL instruction allows the user to push the value(s) of a variable, string, or explicitly DIMensioned matrix onto the run time stack. The value(s) will automatically be popped back into their respective variables upon completion of a pending control structure.

19. Scope of Variables

5. When a DIMensioned string or arithmetic variable is defined as LOCAL, it must be re-DIMensioned. The LOCAL variable is a new variable with a value of zero (or null characters in the case of a string) and no specified DIMension.

Examples:

```
100 Length=5
110 Do
120 Local Length
130 Length = 10
140 Print Length
150 Enddo
160 Print Length
170 End
```

```
>>Run
10
5
```

In this example, the variable Length is assigned a value of 5 on line 100. A control structure is introduced on line 110. The purpose of the DO-ENDDO control structure is to restore the variable Length to its former value after being used as a LOCAL variable. Line 120 creates a new variable, Length. Length has a value of 0 until, on line 130, it is assigned a value of 10. Line 150 (the termination of a Pending control structure) restores the old value of 5 to Length as can be seen from the PRINT instruction on line 160.

The following example demonstrates a more practical use of the LOCAL instruction. Each time before the recursive Procedure .Factorial calls itself, the variable N is saved (on the run time stack) for later use. Each time .Factorial ends and goes back to a previous level, the variable N is recalled (popped) and used in computing Nfactorial (line 1150).

Refer to the chapter on Procedures for additional information on the use of Procedures and automatic local variables.

19. Scope of Variables

>>List

```

100  Rem Program to compute factorials.
110  Long Answer : Integer N,Boogie : Dim Response$(0)
120  Library"factor"
130  Boogie=1
140  While Boogie=1
150    Input"number: ",N
160    If N>49 Or N<1 Then Do
170      @"The number cannot be larger than 49,"
180      @"Nor can it be smaller than 1."
190    Else
200      Call .Factorial (N;Answer)
210      Print N;"! = ";Answer
220    Enddo
230  @
240  Input"Another? ",Response$
250  Rem Y or y says: calculate another factorial.
260  If Response$#"y" And Response$#"Y" Then Boogie=0
270  Endwhile
280  End

```

>>Use 7

>>List

```

1000 Procedure .Factorial (N)
1010  Rem Recursive Procedure for Computing Factorials
1020  Rem SAVE file name: FACTOR
1030  Integer New'n
1040  If N=1 Then Do
1050    Rem initialize Nfactorial and End Procedure
1060    Nfactorial=1
1070  Else
1080    Do
1090      Rem Decrement N and Call Factorial
1100      New'n=N-1
1110      Local N
1120      Call .Factorial (New'n;Nfactorial)
1130    Enddo
1140    Rem Compute Nfactorial on the way back
1150    Nfactorial=N*Nfactorial
1160  Enddo
1170 Endproc (Nfactorial)
1180 End

```

>>Run "Main"

number: 3

3! = 6

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
19. Scope of Variables

Another? Y
number: 55
The number cannot be larger than 49,
Nor can it be smaller than 1.

Another? Y
number: 5
5! = 120

Another? n
280 End

PROCEDURES

A major facit of structured programming is modularization. The implementation of Partitioned memory, Procedures, and a Procedure Library makes Cromemco 32K Basic truly a Structured Basic.

Under 32K Structured Basic, the User Area is divided into 8 Partitions which are numbered zero through seven. Until used, each Partition is a null Partition and occupies no space in the User Area. When Basic is loaded, the user has immediate access to Partition zero. This is normally where execution of a program starts and where the main program is stored.

A Procedure is a section of a Basic Program which is designed to do one complete task or an integral subtask. Procedures may call other Procedures. They may have parameters passed to them by the calling routine and also return parameters back to the calling routine. Procedures may be recursive (call themselves, refer to the LOCAL instruction.)

One or more Procedures may be loaded into each Partition. This may be done manually by the USE and LOAD (or ENTER) instructions, or automatically from a RUNning program by the LIBRARY instruction and Procedure CALLS.

Structured Basic allows the user to build a Procedure Library by means of the Basic Program LIBBUILD. Once this Library has been opened (refer to the LIBRARY instruction), all Procedure CALLs which cannot be resolved by reference to one of the eight Partitions will cause the Library to be searched for the requested Procedure.

If the Procedure is found, it will be loaded into an available Partition (refer to the Procedure CALL instruction).

Unless explicitly requested (by being passed as a parameter or by the use of a Common Storage Area,

20. Procedures

refer to the BEGINCOMMON and ENDCOMMON instructions), variables are local to a Partition. If more than one Procedure resides within a Partition, the variables are global for all Procedures within that one Partition. Thus, if Procedures .AAA and .BBB reside in Partition zero and Procedure .ZZZ resides in Partition seven and the variable Time is defined in Procedure .AAA, the variable Time will be accessible from Procedure .BBB (in the same Partition as its definition) but will not be accessible from .ZZZ (in another Partition) unless it is passed in common or as a parameter.

When parameters are passed to a Procedure (refer to the Procedure CALL instruction, parset1) they are automatically declared as local variables within the called Procedure. This is beneficial for calls to other Procedures within a single Partition or for the implementation of recursive Procedures.

20. Procedures

program: Library Builder

format: Run "LIBBUILD"

The LIBBUILD program allows the user to create and modify a Procedure Library. LIBBUILD will prompt the user for the necessary file references.

Note:

1. A Library may be composed of many different Modules or just one SAVE file. The LIBBUILD Program is required to group a number of different SAVE files into a single Library which can automatically be searched for a needed Procedure. All files requested by the LIBRARY BUILDER must be SAVED Basic Procedures. Each file (or Module) may contain one or more Procedures.

A CALL to a Procedure which is in the Current Library will cause all Procedures in the same Module (SAVE file) to be loaded into one Partition. If only one Procedure is to be loaded into each Partition, then only one Procedure may be SAVED at a time. This will ensure that there is only one Procedure in each Module and therefore only one Procedure in each Partition.

Example:

As an example of creating and building a LIBRARY, let us assume that we have a program which consists of a main program, two Procedures (one of which usually calls the other several times in succession), and three larger Procedures (no two of which need to be in memory at the same time). The two smaller Procedures are named .AA and .BB while the larger ones are .XX, .YY, and .ZZ.

While in Basic, ENTER the two Procedures .AA and .BB into the User Area. Then issue a SAVE command. We will call this SAVE file MOD1. Then, in turn, ENTER and SAVE the three large Procedures. These SAVE files will be named MOD2, MOD3, and MOD4.

Next, LOAD and RUN the LIBBUILD program (supplied on the disk with Cromemco 32K Structured Basic.)

20. Procedures

The first selection from the program menu must be C for Create Procedure Library. The name of the Library may be any legal CDOS file name.

Continue to follow the prompts of LIBBUILD and Add the four SAVE files to the Library which was just created.

If the Library name is specified in a LIBRARY instruction in a Basic program, a CALL to any of the four Procedures will automatically load the Procedure into the highest numbered available Partition (refer to the CALL instruction.)

instruction: Procedure Call

format: [Ln] [CALL] .prname [(parset1; parset2)]

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

.prname is a procedure name

parset1 is an optional set of one or more actual parameters to be passed to the called PROCEDURE. These parameters may include one or more of the following:

mvar an implicitly or explicitly DIMensioned one, two, or three DIMensional matrix. In this list the name of the matrix must be preceded by the word MAT.

svar a string variable.

aexp an arithmetic expression, variable, or constant.

parset2 is an optional set of one or more return parameters whose values are received from the called PROCEDURE. Parset2 is composed of arithmetic variables only.

The CALL instruction transfers control to a PROCEDURE.

Notes:

1. Parset1 is composed of variables which are to be transferred to the PROCEDURE. This list must match the list in the PROCEDURE definition (refer to the PROCEDURE instruction).

20. Procedures

2. Parset 2 contains return variables. This list must match the list in the ENDPROC or EXITPROC instruction (refer to the ENDPROC and EXITPROC instructions).
3. The semicolon (;) is part of parset2. It must precede parset2 but is not to be included if parset2 is not present.
4. Implementation of Procedure Library search:

When a PROCEDURE is referenced by a Basic program, the following locations will be searched until it is found:

1. the Current Partition,
2. all other Partitions, commencing with Partition zero.

If the PROCEDURE is located in one of the Partitions, control will be transferred to the PROCEDURE in that Partition. If 1 and 2 above are not successful, the search will continue with:

3. the Current Library.

If the PROCEDURE is found within the Library, the Module containing the PROCEDURE will be loaded into the highest numbered Available Partition. An Available Partition is one which is neither manually nor automatically locked. Manual locking is invoked by the use of the Lock instruction. Locking occurs automatically when nested calling of PROCEDURES (in other Partitions) takes place. When control is transferred out of a Partition by a PROCEDURE CALL, that Partition is locked. When control is transferred out of a Partition by an ENDPROC, ERRPROC, or EXITPROC instruction, that Partition is unlocked (assuming that no active control structures remain).

If the search is not successful, an error will be generated.

5. The variables in parset1 are automatically declared as local to the CALLED Procedure. This is advantageous when calling another

20. Procedures

Procedure within the same Partition, or when a Procedure CALLs itself (a recursive Procedure).

Example:

```

100  Rem Program to demonstrate the automatic
110  Rem local feature of a Procedure CALL.
120  Rem
130  Dim String'one$(15),String'two$(15)
140  String'one$="AAAAAAAAAAAAAAAAAAAA"
150  String'two$="BBBBBBBBBBBBBBBBBBBB"
160  Print : Print String'one$ : Print String'two$
170  Call .String'proc (String'one$)
180  Print : Print String'one$ : Print String'two$
190  End
200  Rem
210  Rem
1000 Procedure .String'proc (String'two$)
1100  Print : Print String'one$ : Print String'two$
1200  String'two$="CCCCCCCCCCCCCCCC"
1300  Print : Print String'one$ : Print String'two$
1400  Endproc
1500  End

```

```

>>Run
AAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBB

AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA

CCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCC

CCCCCCCCCCCCCCCC
BBBBBBBBBBBBBBBBBBBB
***190 End***

```

The user may wish to refer to the Glossary while reading the following explanation.

This example demonstrates that parameters which are received by Procedure are local to that Procedure. In the main program, the string variables String'one\$ and String'two\$ are DIMensioned, assigned values of 16 A's and 16 B's, and displayed for verification.

When `.String'proc` is CALLED, `String'one$` is passed to the Procedure in which it is referred to as `String'two$`. This is a call by reference. This means that during the execution of the Procedure the two string variables will be equivalent to each other. The two strings are displayed again. `String'one$` maintains its value of A's because all variables which are not used as receiving parameters in a Procedure are global within a Partition. `String'two$` is local to the CALLED procedure because it is a receiving parameter. As such, `String'two$` received the original value of `String'one$` and now it also displays a value of A's.

The next instruction assigns a string of C's to `String'two$`. `String'one$` takes on this same value because the two string variables are equivalent to each other during the execution of the Procedure. The two strings are once again displayed for verification.

Upon the termination of the Procedure (ENDPROC) and a return to the main program, the two string variables are displayed a final time. `String'one$` (global throughout the Partition) maintains its latest value and contains all C's. `String'two$` (which was local to the Procedure) reverts to the value it had at the time the Procedure was CALLED and contains all B's.

statement: Procedure Definition

format: Ln PROCEDURE .prname

Ln PROCEDURE .prname (parset-1)

where:

Ln is a line number

.prname is a Procedure name

parset1 is an optional set of one or more formal parameters which are passed from the calling program. These parameters may include one or more of the following:

mvar a one, two, or three DIMensional matrix which is defined by the calling routine. In this list the name of the matrix must be preceded by the word MAT.

svar a string variable.

avar an arithmetic variable.

The PROCEDURE definition names a PROCEDURE and provides an entry point into the PROCEDURE.

Note:

1. Parset 1 contains receiving variables. These variables must match parset1 in the corresponding Procedure CALL.

20. Procedures

statement: Procedure End

format: Ln ENDPROC

Ln ENDPROC (parset2)

where:

Ln is a line number

parset2 is an optional set of one or more actual return parameters whose values are passed to the calling program. Parset2 can contain arithmetic expressions, variables, and constants.

The ENDPROC statement indicates a logical end of a PROCEDURE and returns control to the calling program.

Note:

1. Parset2 contains expressions whose values are to be returned to the calling program. These must match parset2 in the corresponding Procedure CALL.

statement: Procedure Error End

format: Ln ERRPROC

where:

Ln is a line number

The ERRPROC statement returns control to the calling program in case of a user trapped error. The statement removes all active control structures within the current PROCEDURE and sets the Basic error flag before transferring control to the calling program.

Note:

1. The ERRPROC instruction is useful when an error is detected while one or more WHILE, UNTIL, etc. control structures are pending in a called PROCEDURE. A jump in program logic to an ERRPROC instruction will cause the Basic error flag to be set, the run time stack to be scrubbed back to the PROCEDURE CALL, and control to be transferred to the calling routine.
2. This statement sets SYS(3) = 251. SYS(3) stores the last error encountered in the Basic program.
3. No parameters may be passed by the ERRPROC statement.

statement: Procedure Exit

format: Ln EXITPROC

Ln EXITPROC (parset2)

where:

Ln is a line number.

parset2 is an optional set of one or more actual return parameters whose values are passed to the calling program. Parset2 can contain arithmetic expressions, variables, and constants.

The EXITPROC statement returns control to the calling program when it is not possible for a normal End of Procedure (ENDPROC) to be executed. The statement removes all active control structures within the current PROCEDURE and transfers control to the calling program.

Notes:

1. Parset2 contains expressions whose values are to be returned to the calling program. These must match parset2 in the corresponding PROCEDURE CALL.
2. The EXITPROC instruction is useful when an error or other need for a change in program logic is detected while one or more control structures are pending in a called PROCEDURE. The execution of an EXITPROC statement will cause the runtime stack to be scrubbed back to the last PROCEDURE CALL and control to be transferred to the calling program.

instruction: Clear Partition

format: [Ln] CLEAR aexp

[Ln] CLEAR .prname

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

aexp is an arithmetic expression, variable, or constant which represents a partition number in the range 0-7.

.prname is the name of a procedure.

The CLEAR instruction parallels the SCRatch instruction, but affects only the specified Partition, or the Partition containing the specified PROCEDURE.

Note:

1. The CLEAR instruction overrides the LOCK instruction.

instruction: Select Procedure Library

format: [Ln] LIBRARY

[Ln] LIBRARY svar

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

svar is a string variable or string literal file reference.

When used with a file reference, the LIBRARY instruction causes the specified file to become the Current LIBRARY.

If the file reference is omitted, the LIBRARY instruction closes the Current LIBRARY.

Notes:

1. The Current LIBRARY is the file which will be searched if a requested PROCEDURE is not current in one of the Partitions. Refer to the PROCEDURE CALL instruction.
2. The LIBRARY can be closed by the LIBRARY, CLOSE, SCR, or BYE instructions. If the CLOSE instruction is used, it must not specify a file number (i.e., CLOSE all files).
3. The LIBRARY instruction automatically closes the Current LIBRARY prior to opening a new LIBRARY.
4. The LIBRARY may be a single SAVE file.

command: Use Partition

format: USE aexp

USE .prname

where:

aexp is an arithmetic expression, variable, or constant which represents a partition number in the range 0-7.

.prname is the name of a procedure.

If the USE instruction specifies a Partition number, that Partition will become the Current Partition.

If a PROCEDURE name is specified, then the Partition containing that PROCEDURE will become the Current Partition. If the specified PROCEDURE is not located in one of the Partitions, the Current LIBRARY will be searched. If the PROCEDURE is found within the LIBRARY, the Module containing the procedure will be loaded into the highest numbered Available Partition and this will become the Current Partition.

Note:

1. The USE command allows the user to hand load and EDIT Modules.

instruction: Lock Partition

format: [Ln] LOCK aexp

[Ln] LOCK .prname

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

.prname is the name of a Procedure.

The LOCK instruction causes the specified Partition to become not Available for automatic Module loading. If the LOCK instruction specifies a single PROCEDURE, the entire Partition containing that Module is LOCKed.

Note:

1. The LOCK instruction overrides the automatic lock/unlock feature.

instruction: Unlock Partition

format: [Ln] UNLOCK aexp

[Ln] UNLOCK .prname

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

.prname is the name of a procedure.

The UNLOCK instruction causes the specified Partition to become Available for automatic Module loading. If the UNLOCK instruction specifies a single PROCEDURE, the entire Partition containing that Module is UNLOCKed.

Note:

1. The UNLOCK instruction does not override the automatic Lock feature.

command: Delete Remark Statements

format: DELREM

DELREM L1

DELREM L1,

DELREM L1,L2

where: L1 is an optional line number or line name.

If L1 is omitted, all lines of the program are processed.

If L1 is the only argument and there is no comma following it, then L1 is the only line processed.

If L1 is the only argument and the comma is included, L1 through the last line in the program are processed by the command.

L2 is an optional line number or line name which indicates the last line to be processed. If included, it must be preceded by L1 and a comma.

The DELREM command DEletes all REMark statements which occur in the lines specified by L1 and L2.

Notes:

1. The DEletion of REMark statements from a program will reduce the amount of memory space needed to RUN it and disk space needed to store it. However, it will have virtually no effect on the execution time of the program.
2. Because DEleted REMark statements are not recoverable, the following procedure is recommended for the use of DELREM:
 - a. After a program has been debugged, LIST or SAVE a copy of the program on a disk

21. Program Security

file before using DELREM.

- b. Use DELREM to delete REMark statements.
- c. SAVE a copy of the new, shorter version of the program in a different file. The file extension NSV could be used to indicate No remarks, SaVed file.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
21. Program Security

command: Protect Program Lines

format: NOLIST

NOLIST L1

NOLIST L1,

NOLIST L1,L2

where:

L1 is an optional line number or line name.

If L1 is omitted, all lines of the program are protected.

If L1 is the only argument and there is no comma following it, then L1 is the only line protected.

If L1 is the only argument and the comma is included, L1 through the last line in the program are protected by the command.

L2 is an optional line number or line name which indicates the last line to be protected. If included, it must be preceded by L1 and a comma.

The NOLIST command allows the programmer the option of keeping part or all of a Basic file confidential. After the execution of the NOLIST command, the specified lines cannot be LISTed.

Notes:

1. Because NOLISTed lines can never be LISTed, the following procedure is recommended for the use of NOLIST:
 - a. After a program has been debugged, Save or LIST a copy of the program in a disk file before using NOLIST.
 - b. Use NOLIST to protect those parts of the

21. Program Security

program which are confidential.

- c. SAVE a copy of the NOLISTed version of the program in a different file. One could use the file name extension PUB to indicate that it is a PUBLIC file.
2. This command protects only those lines present at the time the command is executed. If, after the NOLIST command has been executed new lines are entered, they will not be protected from LISTing even if they replace already NOLISTed lines.

BASIC-KSAM

Basic-KSAM is the Keyed Sequential Access Method file management system which is a part of Cromemco 32K Structured Basic.

It was developed primarily for use in applications where large files are involved and fast random access is necessary. Such applications include, but are not limited to, inventory control, reservation systems, library systems, accounts receivable, and bill of materials processings.

Files are always accessed dynamically, i.e., sequential and random access instructions can be intermixed freely. Records are identified by a unique user defined data field within the record called the Primary Key. Additionally any number of fields may be designated as Alternate Keys (sometimes called Secondary Keys) for retrieval purposes. Examples of keys are: part numbers for inventory control, account numbers for billing systems, and customer names for mailing list applications.

Records can be accessed randomly by their Primary Key or by any of their Alternate Keys.

In addition, Basic-KSAM supports sequential movement through the file (forward, backward and static), and random access by partial key or relative record number.

Space is automatically allocated to the file when records are added. Any number of files can be processed simultaneously provided that sufficient memory for buffer storage is available.

If a diskette becomes full during processing, the user program may instruct Basic-KSAM to overflow automatically to additional disks without program interruption.

Basic-KSAM utilizes a look-aside buffering

technique which eliminates unnecessary disk accesses thereby increasing retrieval speed and minimizing head movement, as well as drive and media wear.

A number of utility programs (KSAMUT) and Numeric Sorting Conversion functions (IKEY\$, FKEY\$, KEYI, and KEYF) are also provided as part of Basic-KSAM.

22. Basic-KSAM Discussion

22.2 STRUCTURE OF BASIC-KSAM FILES

Logical Structure

Basic-KSAM files consist of three portions:

- A. The data set
- B. The key set
- C. The header

Data Set

The data set contains the actual data records created by the user. These records are arranged in ascending collating sequence on the basis of their Primary Keys. As records are added or deleted, Basic-KSAM keeps the entire file in this sequence.

Records are organized in Blocks (Data Blocks) which are scattered throughout the available disk space. The number of records per Data Block is derived by dividing the block length by the record length. Logical records may not span Data Blocks so that only an integral number of records may be contained in any one Block. Any remaining bytes are left unused.

Unit of Allocation

Basic-KSAM allocates space to a file in multiples of a page (256 bytes). Each Data Block normally occupies four pages (1K bytes).

Key Set

The key set contains various pointers to the Data Blocks and it is maintained by Basic-KSAM.

Header

The header contains information about the structure and contents of the file such as record length, key length, and key displacement. It is created from information supplied by the user at file-creation time and maintained by Basic-KSAM. It occupies 1K bytes.

22. Basic-KSAM Discussion

Physical Structure

A Basic-KSAM file is organized physically as a collection of one or more disk files. These files may reside on one or more volumes (disks). Each file has its own name which is specified at file creation time. The purpose of having more than one file name associated with a Basic-KSAM file is the creation of a multi-volume file.

Multivolume Files

A Basic-KSAM file may extend to more than one disk. The user specifies the file names to be included in the Basic-KSAM file at file opening, and can increment this number at any time during processing by use of the KADDVOL instruction.

All volumes of a given Basic-KSAM file must be mounted and on-line any time the file is accessed.

22. Basic-KSAM Discussion

22.3 LOGICAL RECORDS AND KEYS

Logical records make up the largest portion of a Basic-KSAM file. Records are added to Data Files by the user, and to Alternate Key files by Basic-KSAM at the request of the user.

All records of a given Data File must be of the same length. Data File record length is user defined and may vary from one byte to one Data Block.

Alternate Key file record length is determined by the criteria outlined in the following section.

Records may not span Data Blocks. This implies that the minimum number of records in a Block is one. A Data Block may not span two volumes but Blocks of the same file may be distributed over several volumes (diskettes). The maximum number of records in a Block is derived by dividing the Block length (default value of 1024 bytes) by the record length, and taking the highest integer which is less than or equal to the quotient.

Unused Block Space

When more than one record can fit in a Data Block, the user may specify at file-creation time that a portion of the block be reserved for later addition of records. Although this may waste space on the disk it improves processing speed greatly in those applications where frequent additions to the file are involved.

Keys

Any field within the record can be used as a means of randomly accessing the record. A field used for that purpose is called a Key. Key fields may be up to 250 bytes in length.

All Keys are stored by Basic-KSAM as strings. If the user needs a numeric variable (Integer, Long or Short Floating Point) as a Key, the Basic-KSAM Numeric Sorting Conversion Instructions (IKEY\$, FKEY\$, KEYI and KEYF) must be used. These will convert numbers to strings which may be sorted by Basic-KSAM and will also convert these strings back

into numeric variables which can be used by Basic. Refer to Chapter 18 for a description of these instructions.

Primary Key

One of the record keys must contain a value which is unique for that record and is used to identify and distinguish it from all other records of the same file. This key is called the Primary Key and is declared as such by the user when creating the file. The length and displacement of the Primary Key must be the same for all records of the same file.

Alternate Keys

All other keys are considered to be Alternate Keys and need not be unique. The length and displacement of an Alternate Key must also be the same for all records of the same file.

Examples of Primary Keys are account numbers, part numbers, and Social Security numbers. Examples of Alternate Keys are zip codes and surnames.

Alternate Keys are used to create pointer (index) files called Alternate Key or Inversion Files. These files provide a means of accessing records by the contents of a field other than the Primary Key.

Key Length

The Primary Key may be from one to 250 bytes long. If Alternate Keys are also used, the sum of the longest Alternate Key plus the Primary Key may not exceed 250 bytes.

Record Retrieval

For random retrieval the user normally specifies the key of the desired record when requesting service from Basic-KSAM.

For example if a customer record must be retrieved and the Primary Key consists of the customer account number, that number must be supplied if

22. Basic-KSAM Discussion

random access is desired. Of course, if sequential retrieval is acceptable the file can be read sequentially (forward or backward) until the desired record is found.

The relative position of a record within the file may also be used for retrieval. This is a number ranging from 0 (for the first record) to the actual number of records in the file minus 1. Once a record has been read in this fashion, it can be updated or deleted as desired.

The user accessing records by relative position must note that the relative position of records will change as records are added or deleted.

22.4 ALTERNATE KEY FILES

An Alternate Key is a field in the record which may be used as an alternate means of retrieving the record.

Any and all fields in the record may be designated as Alternate Keys at any time after the file is created. The difference between the Primary Key and any Alternate Key is that Alternate Key values need not be unique. For example if a file of students exists, the Primary Key may be the student's social security number while the major field of study may be used as an Alternate Key. Since more than one student may have the same major, the Alternate Key value will not be unique but the social security number (Primary Key) will be. Student records may then be accessed in order of their major rather than their social security number, thus eliminating the need for sorting the file in major sequence.

An Alternate Key file (inversion file) is a Basic-KSAM file whose records are arranged in ascending order by the value of the Alternate Key field and contain pointers to corresponding records in the Primary or Data File. Any number of Alternate Files may be created for the same Data file, each corresponding to a different Alternate Key field of the Data file record. An Alternate Key file record is an inversion of the corresponding data file record according to the specified Alternate Key. Since Alternate Key files are Basic-KSAM files they can also be read independently of their associated Data file for inquiry and report purposes. The user cannot, however, add, delete, or update records in the Alternate Key file independently using the KADD, KDEL, and KUPDATE instructions. To add or delete records from an Alternate Key file the appropriate Alternate Key instructions must be used while both the Data and Alternate Key files are open.

There is no restriction that there be a one-to-one correspondence between the Data file records and their corresponding records in the Alternate Key file. Adding or deleting Alternate Key file records is not done automatically upon adding or deleting a Data file record, but is left up to the user. One may therefore create Alternate Key records for only a part of the total Data file.

22. Basic-KSAM Discussion

Similarly a Data file record may be deleted or altered without its corresponding Alternate Key record being affected. If the user wishes to maintain a one-to-one correspondence then a record must be added to every Alternate Key file after a new Data file record is added, and deleted before the Data file record is deleted. If a Data file record is to be modified so that the value of an Alternate Key field will be changed, and the user wishes to maintain a one-to-one correspondence, the affected Alternate Key file record must be deleted first, then added again after the Data file record is updated. This will insure that the latest value of the Alternate Key field is stored in the Alternate Key file and that the Data file record can still be accessed through the Alternate Key. If this is not done, as a result of the selective addition and deletion of Alternate Key file records, it is possible that these records may be left stranded in the sense that they are still part of the Alternate Key file but they cannot be used to access any Primary Data file records. This should be avoided as it defeats the purpose of the Alternate Key file.

Alternate Key file records are made up of two fields. The first field contains the value of the Alternate Key field in the corresponding Primary Data file record. Therefore the length of this field equals the length of the corresponding Alternate Key field in the Data file record. The second field contains the Primary Key of the corresponding Data file record and is equal in length to the Primary Key.

The total length of the Alternate Key File record is the sum of the lengths of the Alternate Key and Primary Key fields.

In the example above, if the Social Security field (Primary Key) of the Data file record is 9 bytes and the major code (Alternate Key) is 2 bytes, each record in the Alternate Key file MAJOR will be 11 bytes long.

Since Alternate Key files are Basic-KSAM files, they must also possess their unique Primary Key. Therefore the entire Alternate Key File record constitutes the Primary (and only) Key of the Alternate Key file. Since the Primary Key of the Data file record is unique, the Primary Key of the

Alternate Key file is also unique.

Taking these facts into consideration the user is never required to supply the record length, Primary Key length and Primary Key displacement when creating an Alternate Key file. All that is needed is the length and displacement in the Data file record of the field to be used as an Alternate Key. The file number of an open Data file is passed to Basic-KSAM and these quantities are calculated as follows:

RECORD LENGTH OF ALTERNATE KEY FILE =
LENGTH OF ALTERNATE KEY FIELD +
LENGTH OF PRIMARY KEY OF DATA FILE RECORD

PRIMARY KEY LENGTH OF ALTERNATE KEY =
RECORD LENGTH OF ALTERNATE KEY

PRIMARY KEY DISPLACEMENT OF ALTERNATE KEY FILE = 0

Quantitative inquiries about the Data file can be made without ever accessing it. If the user keeps a one-to-one correspondence between the Data file and all of its Alternate Key files, questions such as:

How many students are enrolled in Engineering?

can be answered quickly without having to access the Data file. After the Data file has been opened, the Alternate Key file MAJOR is opened, and all records corresponding to major = Engineering are counted. Since the Alternate Key File records are generally much smaller than the corresponding Data file records, Alternate Key files can be scanned very quickly.

Temporary Alternate Key files can be created instead of sorting for report generation purposes. These files can be deleted immediately after the desired report is produced. With a little effort the user can implement inquiry programs to quickly answer questions such as:

How many WHITE MALE students are enrolled in ENGINEERING?

The words in capitals correspond to fields in the Data file record which can be used as Alternate Keys.

22. Basic-KSAM Discussion

Remembering that the Alternate Key file record contains the Primary Key of the Data file record corresponding to it, the Data file can always be accessed directly (without the use of Alternate Key instructions) by using information derived from Alternate Key Files.

Note that Basic-KSAM requires that the Primary File be opened in order to open the Alternate Key File. The Primary file need not be accessed.

22.5 THE CURRENT RECORD POINTER (CRP)

At all times Basic-KSAM maintains a pointer to the current record. Special CRP cases are the beginning of file (BOF) and end of file (EOF). All possible positions of the CRP are described in detail below:

- A. Any successful read, add or update operation will cause the CRP (current record pointer) to point to the successfully processed record.
- B. A BOF (beginning of file) condition occurs whenever the CRP points to the dummy record in front of the first logical record of the file. This is the case immediately after the file is opened. Reading backwards (sequentially) past the first record will also set this condition.
- C. When the EOF (end of file) is reached the CRP (current record pointer) points to the dummy record beyond the last record of the file. This condition will occur when reading sequentially forward past the last record, or when a random instruction is given with a key value greater than any key on file.
- D. A successful deletion will cause the CRP to point to the previous record (in the collating sequence of the Primary Keys) or to the BOF if the first record of the file was deleted.
- E. An unsuccessful read, update or delete operation will position the CRP at the first record which has a key value greater than the one specified in the operation. The CRP will be left pointing to the EOF if the specified key is larger than any on file or if sequential reading past the last logical record was attempted.
- F. An unsuccessful add operation will position the CRP at a record in the file which has the same key as the one specified by the user. (Remember that Primary Keys must be unique so that trying to add a record with a key that already exists will cause an error.) An exception to this rule is the case of an add which was unsuccessful because there was no more room available on the disk. In this instance the CRP points as specified in

22. Basic-KSAM Discussion

paragraph E.

- G. If the CRP is positioned at the BOF and a read previous or read current operation is attempted, the CRP is unchanged. This is also true if the CRP is positioned at EOF and a read current or read next is attempted.
- H. If an I/O error occurs during processing the contents of the CRP is unreliable.

When in doubt as to the position of the CRP, issue a Read Current Record, Primary File (KGETCUR) or retrieve key (KRETRIEVE) instruction. This will return either the current record or a BOF/EOF/File Empty error message.

SUMMARY OF BASIC-KSAM INSTRUCTIONS

<u>INSTRUCTION</u>	<u>DESCRIPTION</u>
Add Record, Primary File	Add a record to a Data file in the proper sequence by Primary Key.
Add Record, Alternate File	Add a record to an Alternate Key file corresponding to the current Primary Data file record.
Add Volume to Existing File	Add a volume to an existing file.
Close File	Close a file.
Create Primary Data File	Initialize and format a Basic-KSAM Primary Data file.
Create Alternate Key File	Initialize and format a Basic-KSAM Alternate Key file.
Delete Record, Primary File	Delete a record from a Primary Data file.
Delete Record, Alternate File	Delete the Alternate Key file record (if any) corresponding to the current Primary Data file record.
Load Record, Primary File	Load (add) a record to a Primary Data file and attempt to preserve the unused space in the block. Usually used when building the file.
Open Alternate File	Open an Alternate Key file for further processing.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

Open Primary File	Open a Primary Data file for further processing.
Read Previous Record, Primary File	Read the previous Primary File record in the collating sequence of the Primary Keys.
Read Current Record, Primary File	Read the current record from the Primary Data file.
Read Next Record, Primary File	Read the next Primary file record in the collating sequence of the Primary Keys.
Read Random Record, Primary File	Read a record randomly from the Primary Data file by Primary Key.
Read Approximate, Primary File	Read the first record in the Primary Data file whose Primary Key is greater than or equal to a given key.
Read Nth Record, Primary File	Read the Nth record of the Primary Data file.
Read Primary Record By Current Alternate Key	Read the Primary Data file record corresponding to the current Alternate record.
Read First Primary Record By Specified Alternate Key	Read the first Primary Data file record which contains a given Alternate Key value.

Read Next Primary
Record, Current
Alternate Key

Read the next
Primary Data file
record which contains
the same Alternate
Key value as the
current Alternate Key
file record.

Retrieve Primary Key,
Current Record

Read the Primary Key
of the current
record.

Update Record,
Primary File

Replace a given
record in a Primary
Data file.

Verify Alternate
Record

Read the Alternate
Key File record (if
any) corresponding to
the current Primary
Data File record.
Check that the
contents of the
Alternate record
correspond to the
Primary record.

DESCRIPTION OF BASIC-KSAM INSTRUCTIONS

Basic-KSAM instructions are logically divided into four groups:

- A. Instructions which treat the entire file as a unit.
 - 1. Create Primary Data File
 - 2. Create Alternate Key File (see section D)
 - 3. Close File
 - 4. Open Primary File
 - 5. Open Alternate File (see section D)
 - 6. Add Volume to Existing File
- B. Sequential access in Primary Key sequence.
 - 1. Read Previous Record, Primary File
 - 2. Read Current Record, Primary File
 - 3. Read Next Record, Primary File
 - 4. Retrieve Primary Key, Current Record
- C. Random access by Primary Key.
 - 1. Read Random Record, Primary File
 - 2. Read Approximate, Primary File
 - 3. Update Record, Primary File
 - 4. Delete Record, Primary File
 - 5. Read Nth Record, Primary File
 - 6. Add Record, Primary File
 - 7. Load Record, Primary File
- D. Alternate Key instructions.
 - 1. Read Primary Record by Current Alternate Key
 - 2. Read First Primary Record by Specified Alternate Key
 - 3. Read Next Primary Record by Current Alternate Key
 - 4. Verify Alternate Record
 - 5. Add Record, Alternate File
 - 6. Delete Record, Alternate File

Instructions in group A, with the exception of the Close instruction, expect the file to be closed. If it is not, an Invalid Request error will occur. The Close instruction expects the file to be open.

As was mentioned earlier, it is possible that at a certain point in time the file may be devoid of

data records. This will happen when the only record on the file is deleted or when the file has just been created. If an Empty File is closed the only acceptable instructions are KOPEN and KALTOPEN. If the Empty File is open the only acceptable instructions are LOAD, ADD, and CLOSE. All other requests will return an EMPTY-FILE error.

When the file becomes empty it does not disappear. The file remains open until closed by the user (KCLOSE).

Since I/O errors are possible with every instruction they will not be explicitly mentioned where error return codes are discussed. This is also true for all system related errors such as File Not Found, etc. Note that the Invalid Request code can be returned by any instruction.

Refer also to Chapter 18, Machine Level Instructions, KSAM Numeric Sorting Conversions. These instructions are necessary to create key fields from numeric variables.

FILE INSTRUCTIONS

Before a file can be accessed it must be created. This is the function of the KCREATE (KALTCREATE) instruction. This instruction does not actually add any data records to the file (this is done through the KADD (KALTADD) and KLOAD instructions), it only preformats certain fixed areas and allocates initial disk space to the file.

After the file is created it exists in an empty state until a record is written to it. Note that a file may be created in one program but actually be built in another. After the KCREATE (KALTCREATE) instruction is processed successfully, the file is automatically closed by Basic-KSAM.

Before an existing file can be processed, it must be opened through the KOPEN (KALTOPEN) instruction. An Invalid Request will be returned if any request is issued to an unopened file.

After processing of a file is completed, the file must be closed by the KCLOSE instruction. It is very important to remember to close a file if records were added, deleted or updated.

Refer also to the Alternate Key Instructions section.

instruction: Create Primary Data File

format: [Ln] KCREATE\prl,pkl\file-ref-1,file-ref-2,...

[Ln] KCREATE\prl,pkl,spc\file-ref-1,file-ref-2,...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

prl Primary Data file record length (not including Primary Key)

pkl Primary Key length (maximum of 250 bytes)

spc unused space per block

file-ref is one or more file references to the Primary Data File. Each of the references include a 1 to 8 character file name and optionally a disk drive specifier and file name extension. The file references may be string variables or string literals (enclosed in quotation marks.)

The KCREATE instruction will initialize and format one or more Basic-KSAM files.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
167	Invalid Request
168	Invalid Create Parameters
169	File Exists

Error Result: The file is not created.

2. If more than one file reference is used, the Basic-KSAM file may span more than one volume (disk). More than one file reference must be

22.6 Basic-KSAM Instructions

used for multi-volume files.

3. An attempt will be made by the KLOAD instruction to preserve the unused space per block. This instruction can be used for building a Data file, while the Kadd instruction is generally used to add records after the initial file set up.
4. Refer also to the Create Alternate Key File (KALTCREATE) instruction.
5. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

```
10  Vol1$ = "A:File1"  
20  Vol2$ = "B:File2"  
30  Vol3$ = ""  
40  Kcreate \50,7\ Vol1$,Vol2$,Vol3$,Vol4$
```

Because Vol3\$ is a null string a 2 volume file (Vol1\$ and Vol2\$) is created.

Instruction: Close File

format: [Ln] KCLOSE\fn\

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

fn Primary Data or Alternate Key file number

The KCLOSE instruction will close and update the status of a Basic-KSAM file. This instruction is valid for both Primary Data and Alternate Key files.

Notes:

1. Errors which may be returned:

-----Error-----

<u>Data</u> <u>File</u>	<u>Alternate</u> <u>File</u>	<u>Meaning</u>
167	183	Invalid Request

Error Result: The file is not closed.

2. This request is normally the last one issued after completion of file processing. Not issuing this instruction will result in data being lost if records were added to or deleted from the file, or if any records were updated. If the file is open it must be closed before a Rename or Erase operation is performed. If this instruction is issued while the file is closed then an Invalid Request code will be returned.
3. The CLOSE instruction will CLOSE all open files (including Basic-KSAM files).

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

instruction: Open Primary File

format: [Ln] KOPEN\pfn\file-ref-1,file-ref-2,...

where:

[Ln] is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

file-ref is one or more file references to the Primary File. Each of the references include a 1 to 8 character file name and optionally a disk drive specifier and file name extension. The file references may be string variables or string literals (enclosed in quotation marks.)

The KOPEN instruction makes an existing file available for further processing.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
167	Invalid Request
173	CDOS Open Failed

Error Result: The file is not opened.

2. The file must exist (it must have been previously created).
3. The number of file references in the KOPEN instruction must be the same as the number of file references in the corresponding KCREATE instruction. More than one file reference must be used for multi-volume files.
4. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

```
10  Vol1$ = "A:File1"  
20  Vol2$ = "B:File2"  
30  Vol3$ = ""  
40  Kopen \1\ Vol1$,Vol2$,Vol3$,Vol4$
```

Because Vol3\$ is a null string a 2 volume file (Vol1\$ and Vol2\$) is opened.

5. Each open Basic-KSAM file uses $512 + 256 * (\text{Sys}(11) + \text{Sys}(12))$ bytes. The values for Sys(11) and Sys(12) are determined at the time the file is created. Using the default values of 4 pages per Key Block and 4 pages per Data Block, each Basic-KSAM file will occupy 2304 decimal bytes.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

instruction: Add Volume To Existing File

format: [Ln] KADDVOL\fn\file-ref

where:

[Ln] is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

fn is a Primary Data or Alternate Key file number.

file-ref is a single file reference to the file which is to be added to the Primary or Alternate file. This includes a 1 to 8 character file name and optionally a disk drive specifier and file name extension. The file reference may be a string variable or string literal (enclosed in quotation marks.)

The KADDVOL instruction allows the user to add an additional volume (disk) to an already created file. This is necessary when the file is in an overflow condition.

Note:

1. This instruction is valid for both Primary Data and Alternate Key files.

SEQUENTIAL ACCESS INSTRUCTIONS

These instructions do not require that a key be supplied by the user. Instead, they use the Current Record Pointer (CRP) which was discussed previously.

The Read Previous (KGETBACK) instruction will decrement the CRP unless it is pointing to the BOF (Beginning of File) in which case it will be left unchanged.

The Read Current (KGETCUR) and Retrieve Key (KRETRIEVE) instructions will not alter the CRP.

The Read Next (KGETFWD) instruction will increment the CRP unless it is pointing to the EOF (End of File) in which case it will leave it unchanged.

Although none of these instructions (except KRETRIEVE) return the Primary Key, it may be obtained by executing a KRETRIEVE instruction following any other instruction.

instruction: Read Previous Record, Primary File

format: [Ln] KGETBACK\pfn\

[Ln] KGETBACK\pfn\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

var-1-n numeric and/or string variable list

The KGETBACK instruction returns the previous record in Primary Key sequence.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
162	Beginning of File
167	Invalid Request

Error Result: No movement of data takes place

2. If the CRP is pointing to the EOF, it is decremented and the last record of the file is returned.

If the CRP is pointing to the first record of the file it is decremented so that it points to the BOF, a Beginning of File error is returned, no data is moved, and the CRP is left pointing to the BOF.

If the CRP is pointing to the BOF, a Beginning of File error is returned.

In all other cases the CRP is decremented and the record to which it is pointing (next lower Primary Key on file) is returned.

instruction: Read Current Record, Primary File

format: [Ln] KGETCUR\pfn\

[Ln] KGETCUR\pfn\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

var1-n numeric and/or string variable list

The KGETCUR instruction returns the current record from the Primary Data file.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
162	Beginning of File
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. The CRP is not changed. If it is pointing to the BOF or EOF then the appropriate error code is returned and no movement of data takes place. In all other cases the record to which the CRP is pointing is moved to the variable list. This instruction is normally used in conjunction with random operations.

22.6 Basic-KSAM Instructions

instruction: Read Next Record, Primary File

format: [Ln] KGETFWD\pfn\

[Ln] KGETFWD\pfn\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

var1-n numeric and/or string variable list

The KGETFWD instruction returns the next record in Primary Key sequence.

Notes:

- Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

- If the CRP is pointing to the BOF, it is incremented and the first record of the file is returned.

If the CRP is pointing to the last record of the file an End of File error will be returned, no movement of data will take place, and the CRP will be incremented so that it points to the EOF.

If the CRP points to the EOF an End of File error will be returned, no movement of data will take place, and the CRP will be left pointing to the EOF.

In all other cases the CRP is incremented and the record to which it is pointing (the next higher Primary Key on file) is returned.

instruction: Retrieve Primary Key, Current Record

format: [Ln] KRETRIEVE\pfn\svar

where:

Ln is an optional line number. If
 Ln is included, the instruction
 is executed at run time.
 Otherwise it is executed
 immediately.

pfn Primary Data file number

svar string variable

The KRETRIEVE instruction returns the Primary Key
from the current record in the Primary Data file.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
162	Beginning of File
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. This instruction is similar to the Read Current instruction. The difference is that only the Primary Key of the current record is returned upon successful completion.
3. The KRETRIEVE instruction does not alter the CRP. It returns the Primary Key from the record to which the CRP is pointing.

RANDOM ACCESS INSTRUCTIONS

These instructions use the key (or record number) rather than the CRP to locate the desired record. On completion, successful or otherwise, the CRP is adjusted according to the rules in the section describing the Current Record Pointer (CRP). This allows sequential processing to follow any random instruction.

The following discussion applies to all random access instructions except the Read Nth Record (KGETREC) instruction:

Basic-KSAM will attempt to locate a record in the Primary Data file whose key matches the specified key.

- a. If such a record exists, KADD or KLOAD operations will fail with an Invalid Key return code (Primary Key must be unique) while all other operations will be successful. The CRP will point to that record. If a Delete instruction is successfully completed the CRP will then be moved back by one record (previous record in the sequence of Primary Keys) since the original record will no longer be there.
- b. If the given key is higher than any key in the Primary Data file, KADD or KLOAD operations will append this record to the end of the file and set the CRP pointing to it. All other operations will fail with an End of File return code and set the CRP pointing to EOF.
- c. If the given key does not exist, and it is less than the highest key on file, the KADD and KLOAD operations will add this record in the correct place according to the sequence of the Primary Keys, and the Read Approximate (KGETAPP) instruction will retrieve the next higher key on file. The CRP will be positioned at that record. All other operations will fail with an Invalid Key return code. The CRP will be pointing to the next higher key on file.

The Read Nth Record (KGETREC) instruction will attempt to locate the record by its sequence number

(in the order of Primary Keys) rather than its key.
If found, the CRP will point to it. If not, an End
of File return code is set and the CRP points to
the EOF.

22.6 Basic-KSAM Instructions

instruction: Read Random Record, Primary File

format: [Ln] KGETKEY\pfn,pkey\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

pkey Primary Key (string variable or literal)

var-1-n numeric and/or string variable list

The KGETKEY instruction will return the record with the specified Primary Key.

Notes:

- Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
163	End of file
164	Invalid Key
167	Invalid Request

Error Result: No movement of data takes place

- If a record exists on file with the given key the operation is successful. The record is moved to the variable list and the CRP is set pointing to it.

If the given key is higher than the highest key on file the operation fails with an End of File return code and the CRP points to the EOF.

If the given key is less than the highest key on file but it is nonexistent the operation fails with an Invalid Key return code. The variable list is not altered and the CRP points to the next record (next higher key).

instruction: Read Approximate, Primary File

format: [Ln] KGETAPP\pfn,pkey\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

pkey Primary Key (string variable or literal)

var-1-n numeric and/or string variable list

The KGETAPP instruction will return the record with the specified Primary Key or the next higher Primary Key.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
163	End of file
167	Invalid Request

Error Result: No movement of data takes place

2. If a record exists on file with the given key or a higher key the operation is successful. The record is moved to the variable list and the CRP is set pointing to it.

If the given key is higher than the highest key on file the operation fails with an End of File return code and the CRP points to the EOF.

3. This is a very useful tool for browsing through a file. The advantage it has over read random is that it will normally fail only if the given key is higher than the highest on file. This means that the exact key need not

22.6 Basic-KSAM Instructions

be known for successful retrieval. A partial key with the lower portion filled with blanks will either retrieve the desired record or the record with the next higher key. Sequential processing normally follows this operation. For example if a group of records exists for which the high order portion of the key is the same, a KGETAPP instruction will retrieve the first of the group and the rest can be accessed sequentially. If the key supplied is higher than any on file and the End of File error occurs, the CRP is positioned at the EOF. Otherwise a record is retrieved and the CRP points to it.

4. The exact Primary Key for the returned record may be obtained by executing a KRETRIEVE instruction following the KGETAPP instruction.

instruction: Update Record, Primary File

format: [Ln] KUPDATE\pfn,pkey\

[Ln] KUPDATE\pfn,pkey\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

pkey Primary Key (string variable or literal)

var1-n numeric and/or string variable list

The KUPDATE instruction will update the record with the specified Primary Key.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
163	End of file
164	Invalid Key
167	Invalid Request

Error Result: No movement of data takes place

2. This operation behaves in the same manner as a read random operation. The difference between the two instructions is the direction of the data movement.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

instruction: Delete Record, Primary File

format: [Ln] KDEL\pfn,pkey\

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

pkey Primary Key (string variable or literal)

The KDEL instruction will delete the record with the specified Primary Key.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
163	End of file
164	Invalid Key
167	Invalid Request

Error Result: No record is deleted

2. If the record does not exist in the Primary Data file, this operation will fail in the same manner as the Read Random instruction, the return codes and CRP settings are the same. If the record exists, it will be physically removed from the file and the space occupied by it will be reclaimed. The CRP will be set pointing to the previous record (next lower key) or to the BOF if the first record on file is deleted. The return code will indicate successful completion. If the only record of the file is deleted, then the file exists in an Empty state until a record is written to it.

instruction: Read Nth Record, Primary File

format: [Ln] KGETREC\pfn,rec\

[Ln] KGETREC\pfn,rec\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

rec record number

var1-n numeric and/or string variable list

The KGETREC instruction will return the record whose number in the sequence of Primary Keys is rec.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
161	File Empty
163	End of File
167	Invalid Request

Error Result: No movement of data takes place

2. For successful retrieval the record number (rec) must be in the range from 0 to the actual number of records in the file minus one. The first record in the file is considered to be record 0 (relative position). If the number is within range the operation is successful. The record is returned and the CRP points to it. If not, the operation fails with an End of File return code and the CRP points to the EOF.

instruction: Add Record, Primary File

format: [Ln] KADD\pfn,pkey\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

pkey Primary Key (string variable or literal)

var1-n numeric and/or string variable list

The KADD instruction will add the record (variable list) to the Primary Data file in sequence by the Primary Key.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
164	Invalid Key
167	Invalid Request
170	Key Set Full
171	No Free Blocks

Error Result: No movement of data takes place

2. The record is inserted between two others which carry the next lower and next higher keys or at the EOF (BOF), if its key is higher (lower) than any key in the file. The CRP points to the record.

If a record with the same key already exists, the operation fails with Invalid Key code and the file is not altered. The CRP is left pointing to that record (which can be retrieved by the Read Current Record (KGETCUR) instruction).

If a free block cannot be found on any of the

disks allocated to the file, the operation will fail with a No Free Blocks return code and the contents of the file will not be altered. The CRP will be left pointing either to the next higher key or to the EOF. A Retrieve Primary Key (KRETRIEVE) instruction will establish the position of the CRP.

The file should either be closed and compacted (see UTILITIES), or the number of volumes should be incremented (KADDVOL). Then the KADD instruction may be issued again.

The KADD instruction will disregard the unused space per block option of the KCREATE instruction, even if specified by the user, and will attempt to fit as many records in a block as the block length permits. To preserve unused space the user should write records to the file using the KLOAD instruction.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

instruction: Load Record, Primary File

format: [Ln] KLOAD\pfn,pkey\var-1,...,var-n

where:

Ln	is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.
pfn	Primary Data file number
pkey	Primary Key (string variable or literal)
var1-n	numeric and/or string variable list

The KLOAD instruction will add the record (variable list) to the Primary File in sequence by the Primary Key.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
164	Invalid Key
167	Invalid Request
170	Key Set Full
171	No Free Blocks

Error Result: No movement of data takes place

2. This instruction has practical use only when adding successively higher keys to the end of the file. This might be the case, for example, when building the file from scratch using records which are in the same sequence as the file being created. In this instance the unused space, specified when the KCREATE instruction was issued, will be preserved. This is the only difference between the KADD and KLOAD instructions. If the records are in different sequence or if this instruction is used to insert records anywhere other than the end of the file, KLOAD behaves in the same manner as the KADD instruction (i.e., it attempts to fill the blocks completely).

ALTERNATE KEY INSTRUCTIONS

Alternate Key instructions allow the user to create and maintain Alternate Key (inversion) files and access the Primary Data file by means of Alternate Keys.

All Alternate Key instructions involve two files, the Primary Data file and one of its Alternate Key files.

The Primary Data File must be open when any Alternate Key instruction is issued and, with the exception of KALTCREATE and KALTOPEN, the Alternate Key file must also be open.

During the execution of an Alternate Key instruction an error can occur while accessing either the Primary Data file or the Alternate Key file. If an error occurs on the Primary Data file record then the return code will range from 161 to 174. If an error occurs on the Alternate Key file then the return code will range from 177 to 190.

As was mentioned earlier, all Basic-KSAM instructions, with the exception of KCREATE, KADD, KLOAD, and KUPDATE, can be issued to an Alternate Key file. In this case Basic-KSAM treats the file as a Primary Data file and if any error occurs then returned error codes range from 161 to 174.

The KALTCUR, KALTFIRST, and KALTFWD instructions do not return the Primary Key. The Primary Key can be obtained after the execution of one of these instructions by executing a KRETRIEVE instruction.

instruction: Create Alternate Key File

format: [Ln] KALTCREATE\pfn,akl\file-ref-1,file-ref-2,...

[Ln] KALTCREATE\pfn,akl,akd\file-ref-1,...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

pfn Primary Data file number

akl Alternate Key length not including the Primary Key (maximum of 250 bytes when combined with Primary Key.)

akd Alternate Key displacement not including the Primary Key. Default value of 0.

file-ref is one or more file references to the Alternate File. Each of the references include a 1 to 8 character file name and optionally a disk drive specifier and file name extension. The file references may be string variables or string literals (enclosed in quotation marks.)

The KALTCREATE instruction will create and format one or more Basic-KSAM Alternate Key Files.

Notes:

1. Errors which may be returned:

<u>Error</u>	<u>Meaning</u>
183	Invalid Request
184	Invalid Create Parameters
185	File Exists

Error Result: The file is not created.

2. If more than one file reference is used, the Basic-KSAM file may span more than one volume (disk). More than one file reference must be used for multi-volume files.
3. The Alternate Key displacement does not include the Primary Key. At this point the Primary Key is transparent to the user and need not be taken into consideration.
4. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

instruction: Open Alternate File

format: [Ln] KALTOPEN\afn,pfn\file-ref-1,file-ref-2,...

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number

pfn Primary Data file number

file-ref is one or more file references to the Alternate File. Each of the references include a 1 to 8 character file name and optionally a disk drive character specifier and file name extension. The file references may be string variables or string literals (enclosed in quotation marks.)

The KALTOPEN instruction makes an existing Alternate Key file available for further processing.

Notes:

1. Errors which may be returned.

<u>Error</u>	<u>Meaning</u>
183	Invalid Request
189	CDOS Open Failed

Error Result: The file is not opened.

2. The file must exist (it must have been previously created).
3. More than one file reference must be used for multi-volume files.
4. A null (empty) string used as a file reference indicates to Basic that all subsequent file references in the list are to be ignored.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

```
10  Vol1$ = "A:File1"  
20  Vol2$ = "B:File2"  
30  Vol3$ = ""  
40  Kopen \1\ Vol1$,Vol2$,Vol3$,Vol4$
```

Because Vol3\$ is a null string a 2 volume file
(Vol1\$ and Vol2\$) are opened.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

instruction: Read Primary Record By Current
Alternate Key

format: [Ln] KALTCUR\afn\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number.

var1-n numeric and/or string variable list

The KALTCUR instruction will read the Primary Data file record specified by the current record in the Alternate Key file.

Notes:

1. Errors which may be returned:

-----Error-----

<u>Data</u> <u>File</u>	<u>Alternate</u> <u>File</u>	<u>Meaning</u>
161	177	File Empty
163	179	End of File
164	-	Invalid Key
-	178	Beginning of File
167	183	Invalid Request

Error Result: No movement of data takes place. The CRP for the Primary Data file follows the rules of the KGETKEY instruction. The CRP of the Alternate Key file remains unchanged.

2. An Invalid Key error code can be returned by this instruction. This error will occur when:
 - a) There is no record in the Primary Data file whose Primary Key matches the Primary Key field on the current record of the Alternate Key file (stranded inversion due to deletion of the Primary file record).

22.6 Basic-KSAM Instructions

- b) A record exists in the Primary Data file with the corresponding Primary Key but its Alternate Key value does not match the value of the alternate field on the current record of the Alternate Key file. (stranded inversion due to modification of the Alternate Key field in the Primary Data file record).

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

instruction: Read First Primary Record By
Specified Alternate Key

format: [Ln] KALTFIRST\afn,akey\

[Ln] KALTFIRST\afn,akey\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number

akey Alternate Key (string variable or literal)

var1-n numeric and/or string variable list

The KALTFIRST instruction will read the first (in Primary Data file sequence) Primary file record specified by the Alternate Key.

Notes:

1. Errors which may be returned:

-----Error-----

<u>Data</u> <u>File</u>	<u>Alternate</u> <u>File</u>	<u>Meaning</u>
161	177	File Empty
163	179	End of File
164	-	Invalid Key
167	183	Invalid Request

Error Result: No movement of data takes place. The CRP for the Primary Data file follows the rules of the KGETKEY instruction. The CRP for the Alternate Key file follows the rules of the KGETAPP instruction.

instruction: Read Next Primary Record By
Current Alternate Key

format: [Ln] KALTFWD\afn\

[Ln] KALTFWD\afn\var-1,...,var-n

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number.

varl-n numeric and/or string variable list

The KALTFWD instruction will read the next (in Primary Data file sequence) Primary Data file record specified by the current record in the Alternate Key file.

Notes:

1. Errors which may be returned:

-----Error-----

<u>Data File</u>	<u>Alternate File</u>	<u>Meaning</u>
161	177	File Empty
163	179	End of File
167	183	Invalid Request

Error Result: No movement of data takes place. The CRP for the Primary Data file follows the rules of the KGETKEY instruction. The CRP for the Alternate Key file follows the rules of the KGETFWD instruction.

2. This instruction will attempt to locate the next Primary Data file record that contains the same Alternate Key field value as the current Alternate Key record. Assume that we have issued a KALTFIRST instruction on an Alternate Key file. Successive executions of KALTFWD instruction will access all logical records of the Primary Data file with the same Alternate Key value.

instruction: Verify Alternate Record

format: [Ln] KALTVER\afn\

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number.

The KALTVER instruction attempts to locate the Alternate Key file record which corresponds to the current Primary Data file record.

Notes:

1. Errors which may be returned:

-----Error-----

<u>Data</u> <u>File</u>	<u>Alternate</u> <u>File</u>	<u>Meaning</u>
161	177	File Empty
163	179	End of File
162	-	Beginning of File
-	180	Invalid Key
167	183	Invalid Request

Error Result: If a corresponding record does not exist on the given Alternate Key file an Invalid Key error will be returned. The CRP for the Data File follows the rules of the KGETKEY instruction. The CRP of the Alternate file follows the rules of the KGETFWD instruction.

instruction: Add Record, Alternate File

format: [Ln] KALTADD\afn\

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number

The KALTADD instruction will write a record in the specified Alternate Key file. This record will correspond to the current record of the Primary Data file.

Notes:

1. Errors which may be returned:

-----Error-----		
<u>Data</u> <u>File</u>	<u>Alternate</u> <u>File</u>	<u>Meaning</u>
161	-	File Empty
163	-	End of File
162	-	Beginning of File
167	183	Invalid Request
-	186	Key Set Full
-	187	No Free Blocks

Error Result: The alternate record is not added. The CRP for the Primary Data file is not changed. The CRP for the Alternate Key file follows the rules of the KADD instruction.

instruction: Delete Record, Alternate File

format: [Ln] KALTDEL\afn\

where:

Ln is an optional line number. If Ln is included, the instruction is executed at run time. Otherwise it is executed immediately.

afn Alternate Key file number.

The KALTDEL instruction will delete a record from the specified Alternate Key file. This record is the one which corresponds to the current record of the Primary Data file.

Notes:

1. Errors which may be returned:

-----Error-----

<u>Data File</u>	<u>Alternate File</u>	<u>Meaning</u>
161	177	File Empty
163	-	End of File
162	-	Beginning of File
-	180	Invalid Key (No Alternate Key record to Delete)
167	183	Invalid Request

Error Result: The record is not deleted. The CRP for the Primary Data file is not changed. The CRP of the Alternate Key file follows the rules of the KDEL instruction.

Example Program - Basic-KSAM File Handling

```

2000 Rem Program to demonstrate the establishment and use
3000 Rem of Basic-KSAM Primary Data and Alternate Key files.
4000 Rem
5000 Rem
6000 Dim Supplier$(6),Part$(0),Class$(0)
7000 Rem create and open Primary Data File
8000 Rem Primary File Layout:
9000 Rem      Primary Key- Part Number
10000 Rem      bytes 1-7 Supplier Name
11000 Rem      8 Shipping Class
12000 Kcreate\8,1\ "Primary"
13000 Kopen\1\ "Primary"
14000 Rem
15000 Rem create and open two Alternate Key Files
16000 Kaltcreate\1,7\ "Supplier"
17000 Kaltopen\2,1\ "Supplier"
18000 Kaltcreate\1,1,7\ "Class"
19000 Kaltopen\3,1\ "Class"
20000 Rem
21000 Rem
22000 Read Part$,Supplier$,Class$
23000 While Part$#"0"
24000 Rem build Primary Data file
25000 Kadd\1,Part$\Supplier$,Class$
26000 Rem put corresponding records in two Alternate
27000 Rem Key files
28000 Kaltadd\2\
29000 Kaltadd\3\
30000 Read Part$,Supplier$,Class$
31000 Endwhile
32000 Rem close files, data base is complete
33000 Kclose\1\ : Kclose\2\ : Kclose\3\
34000 Rem
35000 Rem
36000 Rem Open files for query
37000 Kopen\1\ "Primary"
38000 Kaltopen\3,1\ "Class"
39000 Rem error indicates that there are no more
40000 Rem records with the specified Alternate Key
41000 On Error Goto Done
42000 Rem
43000 Rem get first occurrence in Primary Data file of
44000 Rem record with specified Alternate Key
45000 Kaltfirst\3,"1"\Supplier$,Class$
46000 Rem retrieve Primary Key
47000 Kretrieve\1\Part$
48000 Print Part$,Supplier$,Class$
49000 Rem loop through successive records with specified
50000 Rem Alternate Key

```

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.6 Basic-KSAM Instructions

```
5100 *Loop : Kaltfwd\3\Supplier$,Class$
5200   Kretrieve\1\Part$
5300   Print Part$,Supplier$,Class$
5400   Goto Loop
5500 *Done : Kclose\1\ : Kclose\3\
5600   Rem erase files so the program may be run again
5700   Erase"Primary" : Erase"Supplier" : Erase"Class"
5800   End
5900   Rem
10000   Data"9","Fred T.,""1"
10001   Data"4","Mary Q.,""3"
10002   Data"3","Fred T.,""2"
10003   Data"7","Jane S.,""1"
10004   Data"2","Jane S.,""3"
10005   Data"1","Fred T.,""2"
10006   Data"0","0","0"

>>Run
7           Jane S.           1
9           Fred T.           1
***5800 END***

>>
```

This is a Structured Basic Program which demonstrates some Basic-KSAM file building and accessing techniques.

Line 600 DIMensions string variables. A variable which is DIMensioned as 0 contains one element (numbered 0).

Lines 1200 and 1300 create and open the Primary Data file (named PRIMARY) with a record length of 8 and a Key length of 1. Channel number 1 is used for this file.

Lines 1600-1900 create and open two Alternate Key files. The first (SUPPLIER) relates the supplier name (Alternate Key) to the part number (Primary Key). SUPPLIER uses the file which was opened on channel 1 (PRIMARY) as the Primary Key file and has a Key length of 7 with a displacement (by default) of 0.

Note that the displacement does not include the Primary Key. The second Alternate Key file (CLASS) relates the shipping class (Alternate Key) to the part number (Primary Key). CLASS also uses the file which was opened on channel 1 (PRIMARY) as the

22.6 Basic-KSAM Instructions

Primary Key file and has a Key length of 1 with a displacement of 7.

Lines 2200-3100 READ the data from the DATA statements build the Primary Data and two Alternate Key files. Notice that in line 2500 the Primary Key is not part of the variable list.

Line 4500 searches the Alternate Key file CLASS for the first record containing a shipping class of 1. If one is found, the Primary Data file (PRIMARY) is searched for the corresponding record. The class and supplier name are then read from the Primary File. Notice that the Primary Key has not been returned. The next instruction (line 4700) retrieves the current Primary Key.

The KALTFWD instruction searches for additional records in the Alternate Key file which have the same Alternate Key value.

When no more records with the same Alternate Key value exist, an error is generated, trapped by line 4100, and control is passed to line 5500 (line label Done).

The techniques used in this example can be drawn upon and extended to provide the user with examples of the use of all of the Basic-KSAM instructions. As will become self evident, the power of these instructions will allow the user to accomplish many things which were heretofore impossible in Basic.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.7 Basic-KSAM Error Messages

BASIC-KSAM ERROR CODE SUMMARY

<u>ERROR</u>	<u>PRIMARY DATA FILE</u>	<u>ALTERNATE KEY FILE</u>
FILE EMPTY	161	177
BEGINNING OF FILE	162	178
END OF FILE	163	179
INVALID KEY	164	180
INVALID REQUEST	167	183
INVALID CREATE PARAMETERS	168	184
FILE EXISTS	169	185
KEY SET FULL	170	186
NO FREE BLOCKS	171	187
CDOS OPEN FAILED	173	189
CDOS I/O ERROR	174	190

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.7 Basic-KSAM Error Messages

Error	FILE EMPTY
Error Codes	161 For A Primary Data File 177 For An Alternate Key File
Discussion	This error code is returned when an attempt was made to access a record of an Empty file. A file is Empty if no records have been written to it, or if all its records have been deleted. Provided that an Empty file is open, the only allowable requests are KCLOSE, KLOAD and ADD. If an empty file is closed the only request allowed is KOPEN, RENAME, and ERASE.
Remarks	Upon return from a Create (KCREATE) or Create Alternate (KALTCREATE) request the file is empty and closed. Before records can be written to the file, it must be opened. Assume that we have deleted all the records of a file. This file is still open and empty. If we want to ERASE it we must close it first.
Error	BEGINNING OF FILE (BOF)
Error Codes	162 For A Primary Data File. 178 For An Alternate Key File.
Discussion	An attempt was made to read a record before the first record of the file. This error will be returned by a Read Previous (KGETBACK), Read Current (KGETCUR), or Retrieve (KRETRIEVE) instruction. Assume that the current record pointer (CRP) is pointing to the first record of a file. A Read Previous instruction will result in a BOF error condition. If an Open instruction is followed by Read Current or Read Previous instruction a BOF error will result.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.7 Basic-KSAM Error Messages

Error	END OF FILE (EOF)
Error Codes	163 For A Primary Data File 179 For An Alternate Key File.
Discussion	This error code will be returned when an attempt is made to access a record beyond the last record of the file. The Read Next (KGETFWD), Read Current (KGETCUR), Read Random (KGETKEY), Delete (KDEL), Read Approximate (KGETAPP), Retrieve (KRETRIEVE), Update (KUPDATE) instructions as well as most Alternate Key instructions can return an EOF error message.
Error	INVALID KEY
Error Codes	164 For A Primary Data File 180 For An Alternate Key File
Discussion	This error code can be returned by random access and Alternate Key instructions. When writing a record to a file this error indicates that a record with the same Primary Key already exists. When reading, updating or deleting a record the error indicates that there is no such record on file.
Error	INVALID REQUEST
Error Codes	167 For A Primary Data File 183 For An Alternate Key File
Discussion	This error code can be returned by all Basic-KSAM instructions. An Invalid Request error will be returned if: a) A KOPEN instruction is given while the file is open.

22.7 Basic-KSAM Error Messages

- b) A KCLOSE, sequential, or random instruction is given while the file is closed.
- c) An improper instruction was issued to an Alternate Key file.
- d) An Alternate Key instruction was issued to a Data file.
- e) The Primary Data file and the Alternate Key file were not open when an Alternate Key instruction was given.

Error
Error Codes

INVALID CREATE PARAMETERS
168 For A Primary Data File
184 For An Alternate Key File.

Discussion

The Create (KCREATE) and Create Alternate (KALTCREATE) instructions are the only ones that can return this error code. It means that the given create parameters are not correct.

An attempt is being made to create a file with a key larger than the logical record size, key length plus key displacement larger than the record length, key length equal to zero, logical record length larger than Data Block length, or with unused space per block greater or equal to the maximum number of records per block.

Error	FILE EXISTS
Error Codes	169 For A Primary Data File 185 For An Alternate Key File
Discussion	This error code can be returned by a Create (KCREATE) or Create Alternate (KALTCREATE) instruction. It means that a file with the given file name and file type already exists on the given drive.

Error	KEY SET FULL
Error Codes	170 For A Primary Data File 186 For An Alternate Key File
Discussion	This error can be returned only by a KADD, KLOAD, or Add Alternate (KALTADD) instruction. The maximum number of key blocks is 16. This error will occur because an attempt is made to allocate a key block to a file that already has 16 key blocks. The record causing the problem is not added to the file.

To continue, close the file, compact it using the utility program, and proceed by adding the last record again.

Error	NO FREE BLOCKS
Error Codes	171 For A Primary Data File 187 For An Alternate File
Discussion	This error can be returned only by the Add (KADD), Load (KLOAD), or Add Alternate (KALTADD) instruction. An attempt was made to allocate a new Data Block to the file. No disk space was found on any disk to which the file can extend.

The record which was to be written is not added to the file. If the number of volumes cannot be increased, the file must be closed and then compacted using the utility

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.7 Basic-KSAM Error Messages

program.

Error	CDOS OPEN FAILED
Error Codes	173 For A Primary Data File 189 For An Alternte Key File
Discussion	This error is returned when CDOS is not able to open the file as requested. This can be caused by a full directory. It can also occur because the file already exists.
Error	CDOS I/O ERROR
Error Codes	174 For A Primary Data File 190 For An Alternate Key File
Discussion	CDOS returned an error code when an attempt was made to access the disk.

SUMMARY OF BASIC-KSAM UTILITIES

<u>COMMAND</u>	<u>DESCRIPTION</u>
CHANGE DISKS	Change the disks and initialize the bit maps.
CREATE	Create a Primary Data or Alternate Key file.
ERASE	Remove a Basic-KSAM file from the directory
COMPACT	Compact the file
REORGANIZE	Reorganize the file
STATUS	Print file status
RENAME	Rename a file
COPY COMPACT	Copy the file in compacted form
COPY REORGANIZE	Copy the file in reorganized form

DETAILED DESCRIPTION OF THE BASIC-KSAM UTILITIES

Loading: Type KSAMUT in response to the
 CDOS prompt.

The program will display a list of functions on the console screen. The user can select one function at a time.

If the selected command needs more information before processing, then the user must follow the prompts displayed on the screen by the utility program.

If a file has to be accessed then the file unit, name, and type of the first volume must be provided. If the file extends to more volumes you will be asked to provide file unit, name, type information for all volumes.

All volumes must be mounted.

Any number of requests can be executed on the same or different files of disks without exiting from the program.

Return Codes: All Basic-KSAM and Cromemco
 Disk Operating System (CDOS)
 Error Codes are possible.

If the execution of any request is successful the program responds with "SUCCESSFUL", displays the menu on the screen again, and waits for the next request.

If the request failed then the appropriate message is printed and the program returns to the menu waiting for the next request.

In this discussion, file-ref indicates a file reference of the form:

[X:]FILENAME.XYZ

In the above file reference, X is an optional disk drive specifier. If omitted, the file is assumed to be on the current drive.

Note that all commands and file references given while running KSAMUT must be given in upper case characters.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
22.8 Basic-KSAM Utilities

Request	CHANGE DISKS
Definition	Change the disks and initialize the bit maps.
Discussion	This command must be executed if the user wishes to change disks while running KSAMUT. After the completion of this command, Drive A is the current disk drive.
Request	CREATE
Definition	Create a Basic-KSAM Primary Data or Alternate Key file.
Discussion	This command will create a Basic-KSAM file. The user must follow the prompts and provide the needed information. Also refer to the KCREATE and KALTCREATE instructions.
Request	ERASE
Definition	Remove a Basic-KSAM file from the disk directory.
Discussion	If the file extends to more than one disk then all disks should be mounted. If the operation was successful then, all disk space occupied by this file has become available for other use. The file name has been removed from all the disk directories to which it extended. If the operation failed then the appropriate message is printed.

Request	COMPACT
Definition	Rewrite the original file so that its blocks contain the maximum number of logical records.

Discussion	If the execution of this request is successful, then the new file has blocks containing the maximum number of logical records. The file name(s) remain the same. The new file might have fewer data or key blocks in use, but the disk space occupied by the compacted file will be the same as the original.
------------	--

If the execution of this request fails then the appropriate error message is printed.

Request	REORGANIZE
Definition	Rewrite the original file so that every block contains the initial unused space specified at file-creation time.

Discussion	This command tries to rewrite the given file so that blocks contain if possible the unused space specified at file creation time. The disk space occupied by the new file will be the same as the original file.
------------	---

A compact file will not be reorganized.

NOTE	The COPY COMPACT and COPY REORGANIZE commands are more efficient than the COMPACT and REORGANIZE commands.
------	---

Request Definition	STATUS Print file status (Header Information)
Discussion	The following will be printed: FILE NAME FILE TYPE RECORD LENGTH INITIAL UNUSED SPACE PAGES PER KEY BLOCK PAGES PER DATA BLOCK DATA BASE (OR ALTERNATE FILE) KEY LENGTH KEY DISPLACEMENT VOLUMES SPECIFIED VOLUMES IN USE DATA BLOCKS IN USE FILE SIZE IN RECORDS
Request Definition	RENAME Rename a file
Discussion	This command is similar to the rename command of the operating system. The file to be renamed does not have to be Basic-KSAM file. This command does not need to have all volumes of a Basic-KSAM file mounted to be executed.
Request Definition	COPY and COMPACT Copy the given file so that the blocks of the new file contain the maximum number of records.
Discussion	The successful execution of this request will create a compact copy of the original file. The new file can be created on the same or different disks. The new file does not have to extend to the same number of volumes as the original file. If the new file will extend to disks on which the original file exists then their filenames must be different.

The number of data blocks, key blocks and the disk space occupied by the new file could be less than the original file. If the operation fails the appropriate message is printed.

Request	COPY and REORGANIZE
Definition	Copy the original file so that the blocks of the new file contain the initial unused space specified file-creation time.
Discussion	This request is similar to COPY-COMPACT request. Instead of compacting the blocks, however, it attempts to insert in each block the amount of empty space specified by the user at file creation time. The disk space occupied by the new file might be more than that of the original file. The COPY-COMPACT and COPY-REORGANIZE commands are the only commands by which the free data and key blocks are physically removed from the file.

GLOSSARY

[]

Square brackets are used to indicate an optional quantity. The item enclosed in square brackets may be used, in the position indicated, at the user's discretion.

< >

Angle brackets are used to indicate a key on the console which is depressed by the user (e.g., <RETURN>.)

Argument

An argument is an independent variable, constant, or expression used with a Basic instruction whose value can be specified by the user to instruct Basic to perform a certain task. For example, in the instruction:

Print A, 3, C+7

A, 3, and C+7 are arguments to the Basic instruction PRINT.

ASCII

This acronym stands for American Standards Code for Information Interchange. It is an industry standard used to assign numerical codes (0 through 127) to 128 characters used as letters, numbers, arithmetic operators, various symbols, and control characters. The ASC(X) function will return the ASCII equivalent of any argument. A table of ASCII codes is provided for reference in the Appendix.

Available Partition

An Available Partition is one which is not either manually nor automatically locked. Manual locking is invoked by the use of the LOCK instruction. Locking occurs automatically when nested CALLing of PROCEDURES (in other Partitions) takes place. When control is transferred out of a Partition by a Procedure CALL, that Partition is locked. When control is transferred out of a Partition by an ENDPROC, EXITPROC, or ERRPROC instruction, that Partition is unlocked (assuming that no active control structures remain). The LOCK instruction overrides the automatic lock feature.

Basic Library Editor

An editor which allows the user to create (give a name to) a Library or add, delete, or replace Modules within an existing Library.

Basic Word

A Basic word, commonly called an instruction, is an alphanumeric set of characters which briefly describes the operation to be performed by the computer. Some examples of Basic words are:

- List
- Print
- On Error
- Len
- Stop
- Rnd

Binary Code

Binary code is defined as a code where every code element is either a 0 or a 1. Computer instructions and data for most microcomputers consist of unique, 8 bit binary codes.

Command

A command in Basic is an instruction to the computer which specifies an operation to be performed. In contrast to a Basic statement (see the Statement definition), commands are executed immediately. Commands are used primarily to manipulate or execute a program once the program has been entered. Commands have no line number preceding them.

A powerful feature of Cromemco Structured Basic is the ability to use most commands as statements. As such, they may be given line numbers and included in the body of the program for execution while the program is running.

Control Character

A control character is a non-printing ASCII character which is (usually) used to transmit control signals between a peripheral device and the computer. For example, a CTRL-P entered from the console will cause the system printer to echo all information which is displayed on the console.

Control Structure

Control structures are (sets of) instructions which change the order of execution from the sequential line number order. In structured programming preferred structures are conditional loops and branches, which allow program flow to continue linearly, conditionally repeating or skipping over sections of code.

23. Glossary

Current Library

The Current Library is the file which will be searched if a requested PROCEDURE is not current in one of the Partitions. Refer to the LIBRARY instruction.

Current Partition

That Partition in which execution or Editing is taking place. The contents of the Current Partition will be displayed by the LIST instruction. Refer also to the USE instruction.

Current Program

The current program is any program with which the user is currently interacting. When Structured Basic is entered, no program is current. Should the user enter text to create a new program, this program becomes the current program. If the user calls a SAVED program from system memory, that program becomes the current program. If the user EDITS a program, it remains the current program in its edited form.

Data

The term is used in two ways. Strictly speaking, any information contained within memory or control logic is binary data. Whether this data becomes alphanumeric characters or control information depends upon the program in use.

In the other sense, data is used to refer to numerical or string information. In Basic, this numerical or string information is listed in a file or DATA statement.

Default

With certain Basic instructions, an argument may be optionally added to control a certain function. If no argument is given, the instruction defaults or reverts to a value already programmed into the Basic interpreter. For example, the default values of the arguments for the command:

Renumber

are 10,10 in Cromemco Basic. This default value for RENUMBER will produce automatic line renumbering starting with line 10 and numbering consecutive lines by increments of 10, (e.g., 10, 20, 30, 40...). To change this default value, the Basic word must be followed by an argument. For example, the command:

Renumber 5,5

will provide automatic line RENUMBERing starting with line 5 and continuing by increments of 5 (e.g., 5, 10, 15,...).

Disk Storage

A disk is a computer memory device which is used to store information. Disks are typically used in place of main memory when large amounts of information must be stored. A floppy diskette is similar in appearance to a phonograph record. Most microcomputer systems currently offer disk storage capabilities through either large or mini floppy disks. The floppy and mini floppy terms refer to the two different sizes (8 inch and 5 inch respectively) of the flexible plastic disks used with the disk assemblies.

In addition, Cromemco offers a hard fixed disk with a very large (10 megabyte) storage capacity.

Entry Point

The starting line of a PROCEDURE within a Module. The Entry Point allows Basic to locate a PROCEDURE within a Module or Library.

Expression

An expression is defined as any combination of variables, constants and operators which is evaluated as a single value or logical condition. For instance, in the statement:

```
10 Let Alpha = (Num1*Num2) + (Alpha*Num3)
```

the (Num1*Num2) + (Alpha*Num3) operation, the value of which is assigned to variable Alpha, is interpreted as an expression. In the statement:

```
10 If Alpha = Beta Then Goto 250
```

the logical comparison Alpha = Beta is called an expression and is evaluated to True (=1) or False (=0).

File or Data File

A File defines a group of related information. This information is addressed by means of a File Reference and usually resides on a floppy diskette.

File Name

This is a one to eight character label which is used to refer to a File. Several Files may have the same File Name. These Files may be uniquely identified by the use of a Disk Specifier and/or a File Name Extension.

Firmware

Firmware is the middle ground between hardware and software. This term is generally applied to specific software instructions that have been burned in or programmed into Read Only Memory (ROM).

Floating Point Mode

Floating point mode refers to a method of computer calculation in which the computer keeps track of the decimal point in each number. In Structured Basic, three formats are used to define variables: Integer, Long Floating Point, and Short Floating Point. In the Long Floating Point mode, numerical values are allowed up to 14 digits. In the Short Floating Point mode, numerical values are limited to 6 digits. The default value in Cromemco Basic is the Long Floating Point (LFP) mode.

Hardware

In comparison to firmware and software, hardware represents the actual metal (or hard) elements of a computer system. Items such as printers, terminals, and the computer itself are considered to be hardware.

Integer

An integer is defined as a whole number, positive or negative. The following numbers are examples of integers and non-integers:

<u>INTEGERS</u>	<u>NON-INTEGERS</u>
3	3.14159
10	.66666
-5	2/3

Integer Mode

Integer mode is a format used to define variables in which one or all variables within a given program are set to integer values only.

Interactive

An interactive device is one used to achieve direct person to computer communication, and vice versa. The teletype and CRT terminals are the best known examples of interactive terminals, although many variations are possible.

I/O (Input/Output)

The I/O initials stand for Input and Output. I/O is the transfer of data between the computer system and an external device. Devices such as CRT (Cathode Ray Tube) terminals, TTY (teletypewriter) terminals, and disk drives are examples of devices that accept the input data from the user, another peripheral device, or from the computer memory, and that output data to the computer or user.

Library

A collection of one or more Basic Modules which has been put into the required Basic Library format by the Basic Library Editor.

Line Number

All lines in Basic begin with a line or statement number. For example:

```
10 Print Peaches,Pears
```

includes the statement number 10. Line numbers can be assigned manually or through the AUTOL command and may be any integer from 1 through 99999. All Basic lines have a unique number which may be used to access lines which require modification or

23. Glossary

deletion from the program.

Line Name

A Line Name follows the Line Number and may be used to access the line for EDITing, or to transfer control to the named line.

Matrix

A matrix is an array of numeric variables in a prescribed form. For example, the array:

```

      3  2  0
      1  4  6
     -3  4  5

```

is a matrix with three rows and three columns. A matrix with m rows and n columns is written:

```

a11 a12 a13 ...a1n
a21 a22 a23 ...a2n
.
.
.
am1 am2 am3 ...amn

```

The individual entries in the matrix are called elements or cells. For example the quantity a_{ij} in the above matrix is the element in row i and column j . Subscripts used to indicate elements always denote the row first and the column second. Cromemco Basic permits the user to define one, two, or three dimensional matrices. A two (i.e., M_{ij}) or three (i.e., M_{ijk}) dimensional matrix is commonly called a table. A one dimensional matrix, a matrix with n columns but only one row, is commonly called a list. For example, the matrix:

```

3, -1, 5, -8

```

is a list (or a matrix) with one row and four columns.

23. Glossary

Memory

The computer memory is used to store information, including programs and data, for future use. Microcomputers typically use semiconductor memories, of which the two most common types are random-access memory (RAM) and read-only memory (ROM). From a hardware perspective, memory consists of an array of bistable, individually addressable elements each of which represents a single binary digit. Information can be stored either in main memory, which commonly consists of RAM or ROM, or external storage devices, which include disks, magnetic tape, and magnetic drums.

Module

One or more PROCEDURES which have been saved under one file name using the Basic SAVE command.

Partition

A subdivision of memory while running under Basic. Memory is divided into eight Partitions, numbered zero through seven (0-7). Each Partition may be loaded with one Module.

Peripheral Device

Peripheral devices are units which are used in conjunction with a computer but which are external to the computer. Peripherals refer to devices such as printers, plotters, terminals, disk storage devices, etc., which can be connected to the computer. The computer is assumed to be the central unit and peripherals are merely support devices.

23. Glossary

Procedure

A section of Basic code as delimited by the PROCEDURE and corresponding terminal ENDPROC, EXITPROC, or ERRPROC instructions.

Program

A computer program is a set of instructions arranged into statement lines. The instructions are used to instruct the computer to perform specified operations in a certain order. Programs are designed and written to solve a wide range of problems and are used in applications as varied as process control, data reduction, telephone systems, mathematical analysis, games, and stock market transactions.

PROM

This acronym stands for Programmable Read Only Memory. PROMs consist of an array of memory cells that can be fixed in certain patterns by the application of higher than normal voltages. These memories are said to be non-volatile; that is, when power is withdrawn the programmed pattern remains.

Recently, EPROMs, or Erasable Proms, have appeared and have found industry wide usage. EPROMs may be erased by exposure to ultraviolet light, and then re-programmed. The Cromemco Bytesaver II is designed to program EPROMs.

Protocol

Protocol is a set of conventions on the format and content of messages to be exchanged between two logical devices. Most often, differences in timing account for failure of devices to communicate. For example, a certain signal might, of necessity, be present to enable an I/O request to a microprocessor's protocol. To match a computer to a terminal, one must know the mutual handshake protocol.

23. Glossary

RAM

RAM stands for Random Access Memory, or read-write memory. In contrast to PROMs, read-write memory can be changed as well as being read. Some RAMs (known as dynamic) retain data for only a fraction of a second and must be refreshed constantly to retain data. All RAM is volatile and must have power applied to retain data patterns.

ROM

A ROM is a Read Only Memory device that is used for storing fixed information. This information is burned in, or programmed, at specific locations when the ROM is manufactured. A ROM cannot be written into during operation. Any ROM that can later be altered is a Programmable Read Only Memory (see PROM). ROM family memories, once burned, retain their data regardless of power contingencies.

Sector

A Sector is a subdivision of a track. A track on a large disk is divided into 26 sectors while a small disk track is divided into 18 sectors. Sectors are numbered starting from number one and each sector holds one record or 128 (80H) bytes.

Software

Software is a term used to refer to the programs, languages and procedures used in a computer. For example, the Structured Basic interpreter as well as any Basic programs are identified as software.

Statement or Statement Line

A statement in Basic is an instruction or series of instructions to the computer. A statement is defined as one line in a Basic program which is preceded by a line number. For example:

```
100 Quantity = Number'per'box * Boxes
```

is defined as a statement. Typically, a statement can contain a maximum of 132 characters.

A powerful feature of Cromemco Structured Basic is the ability to use most statements as commands. As such, they may be used without line numbers and executed immediately. This is very useful for debugging programs.

Cromemco Structured Basic also allows more than one instruction on a single statement line as long as adjacent instructions are separated by a colon (:).

String Literal

A string literal (or string) is a sequence of alphanumeric characters, spaces, and special characters. In Structured Basic, string literals must be enclosed within quotation marks. Examples of valid string literals include:

```
"Cromemco Structured Basic"
```

```
"12345"
```

```
"This program prints square roots"
```

The statement:

```
100 Print "Cromemco Structured Basic"
```

will output the string

```
Cromemco Structured Basic
```

23. Glossary

String Variable

A string variable is a variable which may assume the value of a string literal.

Track

A Track is a physically defined circular path which is concentric with the hole in the center of a disk. It is defined by its distance from the center of the disk. With the read/write head of the disk drive located on a given track, data may be read from or written to that track. A large floppy disk has 77 tracks, while a small disk has 40.

User Area

The User Area is the Basic workspace in which a program can be written, EDITed, and RUN. The LIST command displays the contents of the User Area.

Variable

A variable is a quantity that can assume any one of a given set of values. In Structured Basic, variables are defined by a letter (A through Z) followed by any combination of up to 30 letters, numbers, and apostrophies ('). Examples of legal variable names include:

Oranges

Boxes'of'Oranges

Cost'per'box

C0

Variables represent numeric values. In the statement:

20 Portion = 8 + 2

Portion is the variable and 8+2 or 10 is the value assigned to Portion. A new value can be assigned to Portion at a subsequent point in the program.

BASIC ERROR MESSAGES

24.1 Fatal Errors

-----Error-----

Number Message

Meaning

1 Syntax

This error message covers a number of errors which can occur when the user is entering (typing in) a program. For example:

Unmatched parentheses:

A=(B*(C)

Misspelled words:

Pirnt A

Wrong data type: A\$=3*A

Bad punctuation:

Print A(7;2)

Because there is only one message for all these errors, a dollar sign is printed under the line in error at a position approximately indicating the position of the error.

2 Using Syntax

The format string for a Print Using instruction is in error. For example:

Print Using "#.##!!!", 3.2E9
 (only 3 exclamation marks; 4 required)

3 Number of Arguments

A function call requires a different number of arguments than the number passed to it. For example:

Def Fna(X,Y)=X+Y

Print Fna(J)

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
24. Error Messages

- | | | |
|----|-------------------------|---|
| 5 | Illegal Statement | (1) This can be caused by entering a line with a syntax error and then RUNning the program without correcting the line.

(2) In certain systems, certain statements can be declared invalid. For example, POKE might be illegal in a multi-user system. |
| 6 | Print Item Size | An attempt was made to PRINT a single item which required more characters than the current page width. For example:
Set 0,10
Print "Lots of characters" |
| 7 | Too Many Gosubs | Subroutines are nested within subroutines to a depth which exceeds that allowed by Basic. |
| 8 | Expression Too Complex | Too many levels of expressions, too many parentheses or function references. |
| 9 | Return, No Gosub Active | The program has no place to RETURN. This can be caused by deleting a line with a GOSUB statement and then encountering its corresponding RETURN statement. |
| 10 | Next Without For | FOR and NEXT statements must be paired. This error may occur if a line containing a FOR statement is deleted. |

24. Error Messages

- | | | |
|----|--|---|
| 12 | User Function
not Defined | A user function is referenced by the program but has not been defined. If the line containing the function definition (DEF FNS(X)) is deleted, the function is no longer defined. |
| 13 | Invalid
Dimensions
given | Invalid argument(s) in the DIMension statement. For example:
Dim A(-20) a negative number
Dim B(5,5,5,5) too many subscripts
Dim C\$(20000) too large an integer
(> 16382) |
| 14 | Goto or Gosub
non-existent
line | A GOTO or GOSUB statement refers to a line that does not exist. |
| 15 | Subscript
Value(s) | The values assumed by subscripts must be less than those in the DIMension statement. |
| 16 | Number of
Subscripts | The number of subscripts associated with a variable must match the number of subscripts in the DIMension statement. |
| 17 | Duplicate
definition
of label or
function | An attempt has been made to give two different statements the same line label or to give two different functions the same function name. |
| 19 | Use of
undefined
line label | Control is transferred to a line label which does not exist. |

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
24. Error Messages

- | | | |
|----|---|--|
| 20 | Run time
stack improperly
nested | The terminating
instruction of a control
structure does not match
the initial instruction of
that control structure.
For example:
10 While x
20 Enddo |
| 21 | Attempt to go
back to altered
or deleted line | A statement containing
part of a control
structure was EDITed or
DELETED and then an
attempt was made to return
control to that
statement. |
| 22 | DIM would
overflow top
of existing
COMMON | The maximum size of the
Common Storage Area is
defined in the main
program (Partition zero). |
| 23 | Bad Begincommon/
Endcommon
sequence | An ENDCOMMON instruction
was encountered with no
previous corresponding
BEGINCOMMON instruction. |
| 24 | String/numeric
expression
mismatch | A syntax error for an
expression incorrectly
involving both string and
numeric data. For
example:
If Ax\$ = Bp\$ + 7 Then 200 |
| 71 | No such
procedure
available | PROCEDURE name not found
in the current Partition
or any other Partition or
the Current Library (if
open). |

24. Error Messages

72	Bad arguments to a procedure CALL/ENDPROC	The arguments for a Procedure CALL or ENDPROC do not match the arguments in the PROCEDURE def- inition. For example: 10 Call .Xyz (Mat Aa) . . 500 Procedure .Xyz (String\$) . .
73	No free partitions to load procedure/ module into	All eight Partitions are either manually or auto- matically LOCKed.
74	Invalid procedure library	The specified Library was not properly built. The Library must be composed of one or more SAVED files which have been concatenated by the LIBBUILD program.
99	FEATURE NOT IMPLEMENTED	This feature has not been implemented.
101	End of Statement/ End of Line	This is an internal Basic error - please document and mail to Cromemco, Customer Service Dept.
102	Out of Memory	There is not enough memory to store the array (string) or to execute the specified control structure.

24. Error Messages

24.2 User Trappable (Non-Fatal) Errors

-----Error-----

<u>Number</u>	<u>Message</u>	<u>Meaning</u>
128	File not Found	File not found on disk (file not in directory) or the device name is not in the device directory list.
129	Filename	An illegal file name was passed.
130	Invalid Command for Device	A command was given to a device which that device was incapable of performing. For example: a read command given to a line printer.
131	File Already Open	An OPEN command was given to a file which was already OPEN.
132	File Not Open	A read or write was attempted using a file which had not been OPENed.
133	File Number	The file number requested was outside the allowable range. The file number must be greater than 0 and less than or equal to the maximum channel number. The file number can never be greater than 16.
134	Cannot Open File	A message from the device driver (or CDOS). (A non-zero value returned on OPEN.)
135	No File Space	All files in use. The system must have one unused channel to do a LIST, ENTER, SAVE, or LOAD. CDOS only - no more space on disk (or there are 64 directory entries).

24. Error Messages

136	File Mode Error	A read was attempted from a write only file or vice versa.
137	Cannot Create file	An attempt was made to CREATE a file that already exists.
138	File Read: No Data	End of file read, or, for random access only, an attempt to read a portion of the file which had not been written.
139	File Write	A message from CDOS - an attempt was made to write to a write protected disk or an error occurred while writing to the disk.
140	File Position/status	An attempt was made to read a negative file record or record larger than 240K bytes.
141	No Channels Available	All I/O channels in use. (The maximum number of channels is system dependent.)
142	Cannot Close File	The specified file has been erased from the disk or a different disk has been inserted in the drive.
143	KSAM-Invalid Alternate File Request	An Alternate file request was given to a Primary file or the corresponding Primary file was closed or missing or a KALTCREATE or KALTOPEN instruction was given with an invalid Primary file reference.

24. Error Messages

144	KSAM-Key Length	The combined Key length for both the Primary and Alternate Keys is greater than 250 bytes or the Key length is shorter than originally specified (except KGETAPP).
145	KSAM-Record Size	The variable list in a read or write instruction was greater than the specified record size.
146	Not a KSAM file	A Basic-KSAM operation was attempted on a standard (non-KSAM) file.
147	File is KSAM file only	A standard operation was attempted on a Basic-KSAM file.
148	Wrong Number of Volume Names Specified	The number of files specified on a KOPEN or KALTOPEN must match the number of files specified by the corresponding KCREATE or KALTCREATE instruction plus any files which have been added by use of the KADDVOL instruction.

NOTE: THERE IS A DETAILED DESCRIPTION OF ERRORS NUMBERED 161-190 AT THE END OF CHAPTER 22.

161	KSAM-Primary File Empty	An attempt to access a record of an Empty Primary Data File.
162	KSAM-Primary Beginning of File	An attempt to read a record before the first record of the Primary Data file.
163	KSAM-Primary End of File	An attempt to access a record beyond the last record of the Primary Data File.

24. Error Messages

164	KSAM-Primary Invalid Key	The Primary Key already exists (write operations) or does not exist (read, update, and delete operations.)
167	KSAM-Primary Invalid Request	Refer to the detailed descriptions of error codes in the Basic-KSAM chapter of this manual.
168	KSAM-Primary Invalid Create Parameters	The parameters in the KCREATE instruction are incorrect.
169	KSAM-Primary File exists	A file with the same file name and extension already exist on the specified drive.
170	KSAM-Primary Key Set Full	The Primary Key Set is Full. The file must be closed and compacted using the utility program before proceeding.
171	KSAM-Primary No Free Blocks	No more space on disk. Either add another volume (KADDVOL) or close and compact the file (utility program) before proceeding.
173	KSAM-Primary CDOS Open Failed	The Cromemco Disk Operating System could not open the Primary Data file.
174	KSAM-Primary CDOS I/O Error	The Cromemco Disk Operating System could not access the disk as was required.
177	KSAM-Alternate File Empty	An attempt to access a record of an Empty Alternate Key File.
178	KSAM-Alternate Beginning of File	An attempt to read a record before the first record of the Alternate Key file.

24. Error Messages

179	KSAM-Alternate End of File	An attempt to access a record beyond the last record of the Alternate Key File.
180	KSAM-Alternate Invalid Key	The corresponding Primary Record does not exist (read, update, and delete operations.)
183	KSAM-Alternate Invalid Request	Refer to the detailed descriptions of error codes in the Basic-KSAM chapter of this manual.
184	KSAM-Alternate Invalid Create Parameters	The parameters in the KALTCREATE instruction are incorrect.
185	KSAM-Alternate File exists	A file with the same file name and extension already exist on the specified drive.
186	KSAM-Alternate Key Set Full	The Alternate Key Set is Full. The file must be closed and compacted using the utility program before proceeding.
187	KSAM-Alternate No Free Blocks	No more space on disk. Either add another volume (KADDVOL) or close and compact the file (utility program) before proceeding.
189	KSAM-Alternate CDOS Open Failed	The Cromemco Disk Operating System could not open the Alternate Key file.
190	KSAM-Alternate CDOS I/O Error	The Cromemco Disk Operating System could not access the disk as was required.
200	Invalid Hex Number	Hexadecimal numbers must contain only the characters 0 through 9 and A through F.

24. Error Messages

201	Integer Overflow	A value greater than 32767 was assigned to an integer variable.
202	Function Argument Value	A function was called using an illegal argument. For example: Sqr(-2).
203	Invalid Input	An attempt was made to INPUT non-numeric data into a numeric variable.
204	Input	An attempt was made to INPUT more items than were called for in the INPUT instruction.
205	Not Dimensioned	A reference was made to a subscripted variable which had not been DIMensioned.
206	No Data Statement	An attempt was made to READ past the end of the DATA supplied. Either there was a READ with no DATA statement or there were not as many items in the DATA statement as in the READ list.
207	Data Type Mismatch	An attempt was made to READ a numeric value to a string variable or vice versa. For example: 10 Data 5 20 Read A\$
208	Number Size	An attempt was made to assign a value outside of the range 9.99E+62 to 9.99E-65 to a variable.
209	Line Length	A line longer than 132 characters was entered.
210	Input Timeout	See the SET instruction for information about this error.

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
24. Error Messages

- | | | |
|-----|------------------------------------|--|
| 250 | Overflow/
Underflow | A floating point operation produced a number outside of the range $9.99\text{E}+62$ to $9.99\text{E}-65$. For example: $A=1/0$. Or, Integer arithmetic caused results outside of the range -32768 to 32767 . |
| 251 | Errproc return
from a procedure | This error is set when an ERRPROC instruction is executed. |

ASCII CHARACTER CODES

DEC.	HEX	CHAR.	DEC.	HEX	CHAR.	DEC.	HEX	CHAR.
000	00	NUL (CTRL-@)	043	2B	+	086	56	V
001	01	SOH (CTRL-A)	044	2C	,	087	57	W
002	02	STX (CTRL-B)	045	2D	-	088	58	X
003	03	ETX (CTRL-C)	046	2E	.	089	59	Y
004	04	EOT (CTRL-D)	047	2F	/	090	5A	Z
005	05	ENQ (CTRL-E)	048	30	0	091	5B	[
006	06	ACK (CTRL-F)	049	31	1	092	5C	\
007	07	BEL (CTRL-G)	050	32	2	093	5D	]
008	08	BS	051	33	3	094	5E	^
009	09	HT	052	34	4	095	5F	<
010	0A	LF	053	35	5	096	60	'
011	0B	VT	054	36	6	097	61	a
012	0C	FF	055	37	7	098	62	b
013	0D	CR	056	38	8	099	63	c
014	0E	SO (CTRL-N)	057	39	9	100	64	d
015	0F	SI (CTRL-O)	058	3A	:	101	65	e
016	10	DLE (CTRL-P)	059	3B	;	102	66	f
017	11	DC1 (CTRL-Q)	060	3C	<	103	67	g
018	12	DC2 (CTRL-R)	061	3D	=	104	68	h
019	13	DC3 (CTRL-S)	062	3E	>	105	69	i
020	14	DC4 (CTRL-T)	063	3F	?	106	6A	j
021	15	NAK (CTRL-U)	064	40	@	107	6B	k
022	16	SYN (CTRL-V)	065	41	A	108	6C	l
023	17	ETB (CTRL-W)	066	42	B	109	6D	m
024	18	CAN (CTRL-X)	067	43	C	110	6E	n
025	19	EM (CTRL-Y)	068	44	D	111	6F	o
026	1A	SUB (CTRL-Z)	069	45	E	112	70	p
027	1B	Esc (CTRL-[)	070	46	F	113	71	q
028	1C	FS (CTRL-\)	071	47	G	114	72	r
029	1D	GS (CTRL-])	072	48	H	115	73	s
030	1E	RS (CTRL-^)	073	49	I	116	74	t
031	1F	US (CTRL-_)	074	4A	J	117	75	u
032	20	(SPACE)	075	4B	K	118	76	v
033	21	!	076	4C	L	119	77	w
034	22	"	077	4D	M	120	78	x
035	23	#	078	4E	N	121	79	y
036	24	\$	079	4F	O	122	7A	z
037	25	%	080	50	P	123	7B	{
038	26	&	081	51	Q	124	7C	
039	27	'	082	52	R	125	7D	}
040	28	(	083	53	S	126	7E	~
041	29	)	084	54	T	127	7F	DEL
042	2A	*	085	55	U			

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout
ESC=ESCAPE

OVERLAY STRUCTURE

These programs demonstrate the ability of Cromemco's Structured Basic to overlay or have one program overlay (part of) itself with another. This feature is very useful for running large programs on smaller systems.

Space is most efficiently used if the same line numbers are used in all programs which are to be overlaid. In the following example, program One is run and calls in program Two.

Program "One"

```
1   Goto 10
5   Enter "Two"
10  Data 1,2,3,4,5,6,7,8,9
20  Read A,B,C,D,E,F,G,H,I
30  Goto 5
```

Program "Two"

```
10  X=A+B+C+D
20  Y=E+F+G+H+I
30  Z=X+Y
40  Print X,Y,Z
50  End
```

>>Enter "One"

>>Run

10	35	45
50 End		

When program One is entered and RUN, line 1 passes control to line 10. Then at line 20, the DATA from line 10 is READ into variables A through I. Control is then passed to line 5 which calls in the second program. Program Two overlays lines 10, 20, and 30. The line which follows line 5 is line 10. Now, however, line 10 is a line from program Two. The rest of program Two is executed with the results being printed out by line 40.

AUTOMATIC STARTUP AND
PROGRAM EXECUTION FROM CDOS

A very powerful feature of the Cromemco Disk Operating System (CDOS) is the ability to enter directly into an application program when powering up the computer. This is especially useful for the inexperienced user as there is no need to deal with any of the commands which are used to load and execute a program.

If, for example, the user wants to execute the Basic program START.SAV automatically when CDOS is entered, the following steps should be followed:

1. Make sure that there is a copy of the batch command file '@.COM' on DISK A.
2. Save the Basic program you want RUN in a file (in this example we are using START.SAV). The program must be SAVED (not LISTed) in order for this to work. Our program for this example is:

```
100  Rem This is my
105  Rem application program!
110  A=5
120  B=10
130  Print "The answer is: "; A*B
140  End
```

3. Using the editor, create a file named STARTUP.CMD on a disk A. Note that this must be named STARTUP.CMD as this is the file name that CDOS looks for. In this example, the command file should contain the line:

SBASIC START.SAV

Then, when CDOS is entered, the batch command will call Kbasic which will RUN the SAVED program START.SAV.

4. When the computer is turned on and CDOS is entered (you have to hit the carriage return several times), our example will output the following:

```
CDOS version xx.xx  
Copyright 1978 CROMEMCO, INC.
```

```
A.@ STARTUP  
@(Batch) version xx.xx
```

```
A.Sbasic Start.Sav
```

```
CROMEMCO 32K STRUCTURED BASIC version xx.xx  
Copyright (c) 1977, 1979 Cromemco, Inc.
```

```
The answer is: 50
```

```
***140 End***
```

5. If the first program line is a NOESC or ONESC, the user can be prevented from ever leaving the RUNning Basic program.

MIXED MODE ARITHMETIC

There are three types of numbers in Basic, INTEGER, Short Floating Point (SHORT), and Long Floating Point (LONG), (hexadecimal constants are treated as integers). Any arithmetic computation, assignment, INPUT, or READ can be performed using any one or more of these types of numbers.

An assignment, READ, or INPUT will automatically convert the number to the type of the receiving variable. This is the variable which is on the left of the equal sign in an assignment instruction, and in the DATA list in the READ and INPUT instructions.

In general, numbers are converted to other types freely as needed. For example:

Sin(30) works, even though the SIN function must have a LONG argument. The Integer 30 is converted to a Long Floating Point number.

Sys(2.6) works, even though the SYS function must have an INTEGER argument. The Long (or Short) Floating Point number 2.6 is converted to an integer (rounding to 3.0).

Most problems will occur with mixed mode arithmetic involving INTEGER numbers. Remember that all constants without a decimal point and with a value less than 10,000 are stored as INTEGERS.

Examples:

```
>>Short Short'var
>>Integer Integer'var
>>Short'var = 6 : Integer'var = 1
>>Short'var = Integer'var / 3 * Short'var
```

This example will assign a value of 0 to Short'var. This is because Integer'var/3 is evaluated first

(rules of precedence, arithmetic operators). Because both Integer'var and 3 are INTEGERS, Integer'var/3 is evaluated as a 0 using integer arithmetic. Zero times anything, no matter what type, is still 0.

```
>>Short Short'var
>>Integer Integer'var
>>Short'var = 6 : Integer'var = 1
>>Short'var = Short'var * Integer'var / 3
```

This example will assign a value of 2 to Short'var. This time Short'var*Integer'var is evaluated first, and because this is mixed mode arithmetic, the shorter form is converted to the longer form (the value of Integer'var is converted to SHORT). Then Short'var*Integer'var is equivalent to 6.0*1.0 or 6.0. We are left with 6.0/3, mixed mode again.

The Integer 3 (the shorter type) is converted to Short Floating Point (the longer type) and the division is performed. Short'var is assigned a value of 6.0/3.0 or 2.0.

```
>>Integer Integer'var
>>Short Short'var
>>Long Long'var
>>Integer'var = 1 : Short'var = 3 : Long'var = 1
```

If at this point we give the command:

```
>>Long'var = Long'var/Short'var
```

Long'var will be assigned a value of 0.33333333333333 because Long'var/Short'var is evaluated as Long Floating Point (the longer type).

If instead we had given the command:

```
>>Long'var = Integer'var/Short'var
```

Long'var would have been assigned the value of 0.33333300000000 because I/S is evaluated as Short Floating Point (the longer type, but still only 6 digits of accuracy) and then assigned to a Long Floating Point variable (Long'var) with 14 digits of accuracy.

Conversion of both types of floating point numbers to Integer numbers and vice versa does take time. Also Arithmetic, indexing, and subscripting with

floating point numbers is much more time consuming than it is using INTEGERS.

Time can be saved by using INTEGER numbers wherever possible. Where it is not possible, care in precedence ordering can result in significant time savings. If Long'var is type LONG and Integer'var is type INTEGER, the first of the following commands will execute faster than the second one.

```
>>Long'var = Integer'var*Integer'var*Integer'var*Long'var  
>>Long'var = Long'var*Integer'var*Integer'var*Integer'var
```

This is because, until the last multiplication, the first example is using INTEGER arithmetic. The second example uses Long Floating Point arithmetic from the start, because the Long variable is at the left and this is where evaluation of this expression begins.

PATCH SPACE

The only space available for user patches is at hexadecimal locations 103 to 1FF. The user can POKE and save (using CDOS) routines in this area.

AREAS OF USER INTEREST

Note:

Many of these values may be changed by the SET instruction. If necessary, and with care, they may be changed with the POKE instruction. Refer to the Appendix section on Changing the Number of I/O Channels for an example.

<u>Hex Address</u>	<u>No. of Bytes</u>	<u>Description</u>
D00	3	A jump (JP) to the routine where Basic uses the address of the bottom of CDOS to establish the top of user program space.
D03	3	A jump (JP) to a point in Basic equivalent to a request to enter a new program line. Does <u>not</u> destroy program currently in memory.
D06	3	A jump (JP) to a point in Basic equivalent to issuing a SCRatch command. Does not re-size memory.
D09	3	A jump (JP) to a point in Basic equivalent to the point reached via the jump at D00 <u>except</u> that memory sizing and baud rate initialization do not take place. <u>On entry here, HL register pair should contain the address to be used as top of user program space, and the A register should contain the number of file channels to allocate.</u>

D0C	3	An illegal instruction (FF Hex, actually a RST 38H), which CDOS traps and uses to display an error message.
D0F	3	A jump (JP) to the CDOS warm-start at address 0000. Used by the Bye command to return to CDOS.
D12	2	The address pointer used by Basic to locate the beginning of the Device Driver List. The user may use this address to find the DDLlist or may change it to force Basic to use an alternate DDLlist.
D14	2	The address pointer to the beginning of the Error Message. The format of error messages in memory is as follows: 3 Bytes -- Call to error printer 1 Byte -- Error number n Bytes -- ASCII message, terminated by a 00 byte.
D16	1	The maximum number of channels which can be allocated at one time. Each I/O channel occupies 192 bytes of memory.
D17	1	The default number of characters per line. Basic is shipped with this byte = 80 decimal.
D18	1	The number of characters per Tab position. Basic is shipped with this byte = 20 decimal.
D19	1	The character which is used as a rubout, default value is 5FH (underline).

D1A	1	Default type of numeric variables and constants where 1=Integer, 2=Short Floating Point, and 4=Long Floating Point. The default value is 4.
D1B	1	Line delete character. The default value is control-U.
D1C	1	The carriage RETURN delay (number of null characters sent after a RETURN). The default value is 0.
D1D	8	Backspace echo - must terminate with a 0. The default value is backspace, space, backspace, 0.
D25	1	Number of pages per Basic-KSAM Key Block. The default value is 4.
D26	1	Number of pages per Basic-KSAM Data Block. The default value is 4.

ADDING DEVICE DRIVERS TO BASICDeclaring Device Drivers

All I/O devices to be accessed from Basic must be declared in the device driver list. This list starts at location DDList and each entry is 8 bytes long. These entries are of the following form:

byte 0	device name (ASCII)		byte 1
	first letter	second letter	
2	base address of device letter		3
	LSB	MSB	
4	may be used for device dependent information needed by driver		5
6	reserved		7

The device name (bytes 0 and 1) is used for finding the correct device to OPEN. For example, OPEN\2\"\$LP" would instruct Basic to search the device driver list for device name "LP" (the \$ indicates a device instead of a disk file). The driver's starting address occupies the next two bytes. Actual addresses and equated value may vary depending on which version of Basic is being used. The following two bytes are reserved for use by the driver. Their function(s) are programmer defined. This device driver list can be placed in PROM. These two bytes are thus not intended as a temporary storage area.

A typical use of one of these bytes might be to hold the actual device address to be accessed. This would, for example, allow a general purpose TU-ART driver to drive several ports each dedicated to a different device.

The last two bytes of each table entry are reserved

for future use. The table itself is terminated by the first null driver name (Hex 00 in the first character of the name).

Rules for Device Drivers

The actual device drivers must follow certain prescribed rules. In particular, the first 16 byte locations in the driver must contain address pointers to the various routines (some of which are required) in a Basic driver. The 16 bytes define the addresses of 8 different routines as shown in the following table:

	(LSB)	(MSB)
Byte 0	address of Open routine	Byte 1
2	address of Close routine	3
4	address of Set Status routine	5
6	address of Get Status routine	7
8	address of Putc routine	9
10	address of Getc routine	11
12	reserved (should be 0)	13
14	reserved (should be 0)	15

If any of the routines noted above are not defined for a given driver, the corresponding address field should be set to 0000H. However, if a routine is (or must be, see below) defined but does not do anything, the corresponding address must contain the address of an Xor A, Ret instruction sequence.

A description of what must be accomplished by the various routines follows. Parameters which may (or must) be passed between Basic and the driver routines are always passed in registers, or in the File Control Block (EFCB).

Most of the information needed by the driver subroutines may be obtained from (or saved in) the File Control Block (EFCB).

Register usage is covered in the description of each command routine below:

OPEN: This routine should perform any processing required to initialize the device. For example, a line printer driver would typically issue a form feed on open.

On Entry: (A') = no. of parameters
(0,1,or 2)
(IY) = ADDR of EFCB
0, 1, or 2 parameters are passed to the routine in locations EFCBP1 and EFCBP2 of the EFCB.

On Return: A not equal to 0 says can't open. This is a user-trappable run time error.

CLOSE: This routine should perform processing necessary to shut down the device. For example, a paper tape punch driver might punch out a trailer piece of tape.

On Entry: (IY) = ADDR of EFCB

On Return: A not equal to 0 says can't close. This is a user-trappable run time error.

PUTC:

GETC: These routines perform byte-by-byte transfers to and from the device. Devices requiring a buffer may use the buffer (and/or extended buffer) in the EFCB.

On Entry: (IY) = ADDR of EFCB
(A) is character to be written (PUTC only)

On Return: (A) used for character read (GETC only)

SSTAT: (SPOS)

This routine is used to set the status of the device. For example, a DAZZLER driver might use this to set the X/Y screen position.

On Entry: (A') = no. of parameters (0,1,or 2)
(IY) = ADDR of EFCB
0, 1, or 2 parameters are passed to the routine in locations EFCBP3 and EFCBP4 of the EFCB.

On Return: A not equal to 0 says invalid status request. This is a user-trappable run time error.

GSTAT: (GPOS)

On Entry: (IY) = ADDR of EFCB
(A') = which status is requested

On Exit: (HL) contains appropriate status value. A not equal to 0 sets a user-trappable runtime error.

NOTE: Registers IX and IY cannot be changed by any of these routines.

The EFCB referred to has the following format:

EFFCB	:	DS 1	;0=not in use, 1=in use
EFCBDA	:	DS 2	;Device Driver Address, ;bytes 2&3 From DDLIST
EFCBDD	:	DS 2	;Device Dependent info, ;bytes 4&5 from DDLIST
EFCBP1	:	DS 2	;\These are the two optional ;parameters
EFCBP2	:	DS 2	;/P1 & P2 as used in the ;"Open" statement
EFCBP3	:	DS 2	;\These are the 2 optional ;parameters P1 & P2
EFCBP4	:	DS 2	;/used in PUT, GET, PRINT, ;and INPUT statements
BUFFER	:	DS 179	;Available to user for ;accumulating individually ;passed bytes into a buffer

25. Appendix

When the proper DDLIST entries have been made as shown, operation of the driver is as follows:

Open\1,A,B,\"\$DR"	Calls the Open routine of driver "DR". EFCBP1 = A, EFCBP2 = B A' register = 2, IY= ADDR of EFCB If Open is not needed, it must consist of an Xor A, Ret sequence.
Print\1,X,Y\"Hello"	The Set Status routine of "DR" is called with EFCBP3 = X, EFCBP4 = Y (X and Y are converted to integers first). A' = 2, IY = ADDR of EFCB An Xor A, Ret sequence should be executed at the completion of Set Status. Next, the PUTC routine of "DR" is called 7 times, once for each character to be transmitted plus once with a Carriage RETURN and once with a LINE FEED. A = character, IY = ADDR of EFCB
Input\1\A\$	Set Status routine of "DR" is called with EFCBP3 = %FFFF%, EFCBP4 = %FFFF%, A' = 0, IY = ADDR of EFCB An Xor A, Ret sequence should be executed. Next, the GETC routine of "DR" is called repeatedly, expecting one byte to be returned in A. This continues until a terminator (CR,LF,FF,NULL) is transmitted, or until more than 132 characters have been transmitted.
A = Iostat(1,n) where 0 ≤ n ≤ 255	The Get Status routine of "DR" is called. A' contains n, the requested status parameter and IY = ADDR of EFCB. The status value to be returned should be placed in HL. An Xor A, Ret sequence should be executed last.
Close\1\	The CLOSE routine of "DR" is called. If CLOSE is not needed, an Xor A, Ret sequence must be provided.

Note that if PUT and GET were used in the above calls instead of PRINT and INPUT, binary bytes

25. Appendix

would be transmitted according to variable type and no carriage control information would be sent. I.E., PUT\1\V would transmit two bytes if V is an Integer, four bytes if V is Short Floating Point, and eight bytes if V is Long Floating Point. No carriage RETURN or LINE FEED is sent. Also note that Print\1\"A\",\"B\" will not transmit the comma but will send the proper number of spaces to place \"B\" in the next tab field.

DDLIST (DEVICE DRIVER LIST)

<u>ADDR</u>	<u>CODE</u>				
					; NOTE: THE ADDRESSES SHOWN IN THIS LISTING
					; MAY VARY FROM VERSION TO VERSION, but the
					; format is identical. The actual starting
					; address may be found in locations 0D12,
					; and 0D13.
					DDLIST:
4B6C	53	DDCNLSL:	DB	'SY'	; 'SYSTEM' = CONSOLE
4B6D	59				
4B6E	E84A		DW	DRTUART	
4B70	00		DB	0,0	
4B71	00				
4B72	00		DB	0,0	
4B73	00				
4B74	54		DB	'T5'	; 2ND TUART PORT
4B75	35				
4B76	E84A		DW	DRTUART	
4B78	50		DB	50H,50H	; BOTH BYTES GET ADDR
					; OF 2ND TUART
4B79	50				
4B7A	0000		DW	0	; RESERVED BY SYSTEM
					; CDOS SYSTEM PUNCH READER, AND LIST DRIVERS
4B7C	50		DB	'PU'	
4B7D	55				
4B7E	9C4A		DW	DRCDPU	; CDOS PUNCH DRIVER
4B80	0000		DW	0	
4B82	0000		DW	0	
4B84	52		DB	'RD'	
4B85	44				
4B86	B54A		DW	DRCDRD	; CDOS READER
4B88	0000		DW	0	
4B8A	0000		DW	0	
4B8C	4C		DB	'LP'	
4B8D	50				
4B8E	CA4A		DW	DRCDLP	; CDOS LIST DRIVER
4B90	0000		DW	0	
4B92	0000		DW	0	
4B94	00	DDEND:	DB	0	; END OF DDLIST-NOTE:
					IF DDLIST IS
					EXTENDED, THIS NULL
					should be overwritten
					and then placed at
					end of the new DDLIST
					; RESERVE SPACE FOR
					; MORE DRIVERS
	4B95		DEFS	64	

CDOS PUNCH, READER, AND LIST DRIVER INTERFACES

<u>ADDR</u>	<u>CODE</u>	; SAMPLE DEVICE DRIVERS		
		;		
		; PUNCH		
		;		
		DRCDFPU:		
4A9C	E64A	DW	\$DUMMY	
4A9E	E64A	DW	\$DUMMY	
4AA0	E64A	DW	\$DUMMY	
4AA2	E64A	DW	\$DUMMY	
4AA4	AC4A	DW	\$PUPC	;PUNCH OUT CHARACTER
4AA6	0000	DW	0	
4AA8	0000	DW	0	
4AAA	0000	DW	0	
		;		
4AAC	5F	\$PUPC:	LD	E,A ;GET CHAR TO E REG AS
				;CDOS EXPECTS
4AAD	0E04		LD	C,4 ;CDOS PUNCH ENTRY
		\$PULPJ:		
4AAF	F5	PUSH	AF	
4AB0	CD0500	CALL	CDOS	
4AB3	F1	POP	AF	
4ABF	C9	RET		
		;		
		;		
		DRCDFRD:		
4AB5	E64A	DW	\$DUMMY	
4AB7	E64A	DW	\$DUMMY	
4AB9	E64A	DW	\$DUMMY	
4ABB	E64A	DW	\$DUMMY	
4ABD	0000	DW	0	
4ABF	054A	DW	\$RDGC	;RDR GET CHAR ROUTINE
4AC1	0000	DW	0	
4AC3	0000	DW	0	
		;		
		\$RDGC:		
4AC5	0E03	LD	C,3	;CDOS READER GETC
				;ENTRY PARM
4AC7	C30500	JP	CDOS	;READY TO GO... CHAR
				;RTND IN A
		DRCDFLP:		
4ACA	DF4A	DW	\$LPOP	;OPEN ROUTINE
4ACC	E64A	DW	\$DUMMY	
4ACE	E64A	DW	\$DUMMY	
4AD0	E64A	DW	\$DUMMY	
4AD2	DA4A	DW	\$LPPC	;AND LP PUTC ROUTINE
4AD4	0000	DW	0	
4AD6	0000	DW	0	
4AD8	0000	DW	0	
		\$LPPC:		

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL

25. Appendix

4ADA	54		LD	E,A	;MOVE CHAR TO E
4ADB	0E05		LD	C,5	;CDOS LIST WRITE
					;ENTRY PARM
4ADD	18D0		JR	\$PULPJ	
		\$LPOP			
4ADF	3E0C		LD	A,ASCFF	;ISSUE A FORM FEED ON
					;OPENING FILE
4AE1	CDDA4A		CALL	\$LPPC	
4AE4	AF		XOR	A	
4AE5	C9		RET		;RET WITH 0 STATUS
		\$DUMMY:			
4AE6	AF		XOR	A	
4AE7	C9		RET		

ACCESSORY I/O DRIVERS

The DDLIST contains five drivers which may be accessed by the user. Basic treats these drivers as files. They must therefore be OPENed and CLOSED, but may not be CREATED (because they already exist).

Line Printer Driver

The system line printer can be accessed by OPENing a file named \$LP. This driver uses the CDOS system call to output data to the line printer. INPUT and GET instructions must not be used with this file.

Punch Driver

The system punch may be accessed by OPENing a file named \$PU. This driver uses the CDOS system call to output data to the punch. INPUT and GET instructions must not be used with this file.

Reader Driver

The system reader may be accessed by OPENing a file named \$RD. This driver uses the CDOS system call to input data from the reader. PRINT and PUT instructions must not be used with this file.

Console Driver

The system console may be accessed by OPENing a file named \$SY. This driver bypasses CDOS and allows the user to input and output data from/to the console directly by using port 0 (UART status) and port 1 (UART data).

TU-ART Driver

There is a general purpose TU-ART serial I/O port driver in the DDLIST called T5. It assumes a TU-ART addressed at port 50H and may be used for any serial device provided the baud rate is initialized prior to its use with an OUT instruction.

Example:

```
10 Out %50%,4 : Rem Set Tu-art to 300 baud
```

```
20 Open\2\"$T5" : Rem sets up device #2 for serial I/O
```

The driver for the system console at port 0 is labeled "\$SY". To switch between the console and a second terminal:

```
20 Open\1\"$SY"
```

```
30 Open\2\"$T5"
```

This will allow you to later say Print\A\"MESSAGE", where A can be changed via software to be 1 for the console and 2 for another terminal.

CHANGING THE NUMBER OF I/O CHANNELS

Basic carries 8 I/O channels in addition to the console. One I/O channel is needed for each file which is OPENed at the same time. Each I/O channel occupies 192 bytes of memory.

If you wish to change the number of channels in use, type the following commands while in Basic:

Disk System

Poke%D16%,X

A=Usr(%D00%,0)

where X is the number of I/O channels desired in addition to the console. The POKE command puts the number of channels desired into the proper location and the USR function reinitializes Basic.

This new version of Basic can be Saved by returning to CDOS and typing:

>>Bye

A.Save Sbasic1.Com 140

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
Index

I N D E X

A

ABS, 190
Absolute Value, 190
Accessory I/O Drivers, 401
Add Record, Alternate File (Basic-KSAM), 335
Add Record, Primary File (Basic-KSAM), 355
Add Volume To Existing File (Basic-KSAM), 307
Address Of A Variable, 239
ADR, 239
Alternate Key (Basic-KSAM), 288, 290
And Boolean Operator, 45
Arctangent, 204
Areas Of User Interest, 389
Argument, 353
Arithmetic Function, 187
Arithmetic Operators, 39, 50
Arithmetic Variable, 52
ASC, 211
ASCII, 163, 353
ASCII Hex Representation, 214
ASCII Table, 381
ASCII Value of a Character, 211
Assignment Instruction, 97
Assignment Operator, 40, 51, 97
ATN, 204
AUTOL, 60, 69
Automatic Line Numbering, 69
Automatic Startup, 383
Available Partition, 354

B

Basic Library Editor, 354
Basic Word, 354
Basic-KSAM, 283
Basic-KSAM Error Codes, 340
Basic-KSAM Numeric Sorting Conversions, 249
Basic-KSAM Utilities, 346
BEGINCOMMON, 253
BINADD, 191
BINAND, 191
Binary Code, 354
Binary Operations, 191
BINOR, 191
BINSUB, 191
BINXOR, 191
Blank Character, 15
Boolean Operator, 45
BYE, 70

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL

Index

C

CALL, 265
Call a User Program, 245
Chaining, 63
CHANGE, 74
Change Disks (Basic-KSAM Utilities), 349
Changing the Number of I/O Channels, 403
Character, 212
CHR\$, 212
CLEAR, 273
Clear Partition, 273
CLOSE, 163, 172
Close File, 172, 304
Command, 16, 355
Command Mode, 50
COMMON, 251
Common Storage Area Method I, 251
Common Storage Area Method II, 253
Compact (Basic-KSAM Utilities), 350
CON, 113
Console Driver, 401
Constant Format, 25
Continue Program Execution, 113
Control Character, 355
Control Structure, 355
Copy and Compact (Basic-KSAM Utility), 351
Copy and Reorganize (Basic-KSAM Utility), 352
COS, 205
Cosine, 205
Create (Basic-KSAM Utility), 349
CREATE, 162, 169
Create Alternate Key File (Basic-KSAM), 325
Create File, 169
Create Primary Data File (Basic-KSAM), 302
CRP, 294
Current Library, 356
Current Partition, 356
Current Program, 356
Current Record Pointer (Basic-KSAM), 294

D

DATA, 147, 356
Data Files, 161
DDLIST, 398
DEF FNS, 208
Default, 357
Define Local Variable, 256
DEG, 101
Degree Mode, 101
DELETE, 71
Delete Record, Alternate File (Basic-KSAM), 336
Delete Record, Primary File (Basic-KSAM), 319

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
Index

Delete Remark Statements, 279
Delete Statement Lines, 71
DELREM, 279
Device Driver List, 398
Device Drivers, 392
DIM, 102
Dimension, 102
Dimensioning String Variable, 32
DIR, 61, 73
Directory, 73
Disable Echo, 225
Disable Escape, 227
Disable Trace Option, 94
Disk Drive, 223
Disk Storage, 357
DO, 127
DSK, 223
Dump Utility (CDOS), 168

E
ECHO, 224
EDIT, 74
Editing, 57
ELSE, 127
Enable Echo, 224
Enable Escape, 226
Enable Trace Option, 93
END, 114
End Program Execution, 114
ENDCOMMON, 253
ENDPROC, 270
ENDWHILE, 132
ENTER, 78
Enter File, 78
Entering from a Disk File, 62
Entry Point, 358
Erase (Basic-KSAM Utility), 349
ERASE, 173
Erase File, 173
Error Messages, 369
ERRPROC, 271
ESC, 226
Example Program, 64, 122, 128, 133, 184, 217, 247, 255, 258, 267, 337
Examples, 49
Execution Mode, 54
EXITPROC, 272
EXP, 193
Expand, 189, 213
Expand String, 213
Exponent, 193
Expression, 358

Index

F

Fatal Errors, 369
Fields, 162
File Name, 358
File or Data File, 358
File Pointer, 166
FIND, 74
Firmware, 359
FKEY\$, 249
Floating Point Constant, 25
Floating Point Mode, 359
Floppy Diskette, 162
FOR-NEXT, 115
FRA, 194
Fractional Portion, 194
FRE, 228
Free Space, 228
Functions, 187

G

GET, 165, 182, 184
Get Record, 182
Global Variable, 253, 262
GOSUB, 129
GOSUB-RETURN, 119
GOSUB-RETRY, 121
GOTO, 58, 123, 129

H

Hardware, 359
HEX\$, 214
Hexadecimal, 23
Hexadecimal Constant, 26

I

IF-THEN, 125
IF-THEN-ELSE, 127
IKEY\$, 249
Immediate Mode, 50
IMODE, 104
Implied Let, 98
INP, 240
INPUT, 137, 164, 166, 178
Input From I/O Port, 240
Input (from the console), 137
Input/Output, 360
Instruction Syntax, 15
INT, 195
Integer, 21, 105, 359

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL

Index

Integer Constant, 25
Integer Mode, 104, 360
Integer Portion, 195
Integer Random Number Generator, 196
Integer Variable, 105
Interactive, 360
Internal Machine Representation, 163
I/O, 360
I/O Channels, 403
I/O Drivers, 401
I/O Status, 229
IOSTAT, 229
IRN, 196

K

KADD, 321
KADDVOL, 307
KALTADD, 335
KALTCREATE, 325
KALTCUR, 329
KALTDEL, 336
KALTFIRST, 331
KALTFWD, 332
KALTOPEN, 327
KALTVER, 334
KCLOSE, 304
KCREATE, 302
KDEL, 319
Key (Basic-KSAM), 287, 288, 290
KEYF, 249
KEYI, 249
KGETAPP, 316
KGETBACK, 309
KGETCUR, 310
KGETFWD, 311
KGETKEY, 315
KGETREC, 320
KLOAD, 323
KOPEN, 305
KRETRIEVE, 312
KUPDATE, 318

L

LEN, 215
Length of String, 215
LET, 97
LFMODE, 106
LIBBUILD, 263
LIBRARY, 274, 360
Library Builder, 263
Line Name, 17, 361

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
Index

Line Number, 54, 360
Line Printer Driver, 401
LIST, 55, 79
List Current Program, 79
List Variables, 81
Listing to a Disk File, 61
LOAD, 63, 83
Load Record, Primary File (Basic-KSAM), 323
Local, 262, 267
LOCAL, 256
LOCK, 276
Lock Partition, 276
LOG, 197
Logarithm, 197
LONG, 108
Long Floating Point, 23
Long Floating Point Mode, 106
Long Floating Point Variable, 30
Long Variable, 108
LVAR, 81

M

MAT, 99
Matrix, 30, 361
Matrix Initialization, 99
MAX, 198
Maximum Value, 198
Memory, 362
MIN, 199
Minimum Value, 199
Mixed Mode Arithmetic, 385
Module, 362
Multiple Instruction Line, 18
Multivolume Basic-KSAM Files, 286

N

NEXT, 115
NOECHO, 225
NOESC, 227
NOLIST, 281
Non-Fatal Errors, 374
Not Boolean Operator, 46
NTRACE, 94
Numeric Internal Machine Representation, 21
Numeric Sorting Conversions (Basic-KSAM), 249
Numeric Variable, 29

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
Index

O

ON ERROR, 231
On Error Transfer Control, 231
ON ESC, 233
On Escape Transfer Control, 233
ON-GOSUB, 129
ON-GOTO, 129
OPEN, 163, 170
Open Alternate File (Basic-KSAM), 327
Open File, 170
Open Primary File (Basic-KSAM), 305
Operator, 39
Or Boolean Operator, 45
OUT, 242
Output To I/O Port, 242
Overlay Structure, 382

P

Partition, 261, 362
Patch Space, 388
PEEK, 243
Peek At Memory, 243
Peripheral Device, 362
POKE, 244
Poke Into Memory, 244
POS, 216
Position of Substring, 216
Primary Key, 288
PRINT, 50, 140, 163, 166, 175
Print (to the console), 140
PRINT USING, 149
Printer, 56
PROCEDURE, 261, 269, 363
Procedure Call, 265
Procedure Definition, 269
Procedure End, 270
Procedure Error End, 271
Procedure Exit, 272
Program, 54, 363
Programmer Defined Function, 188, 208
PROM, 363
Prompt (>>), 16, 50
Protect Program Lines, 281
Protocol, 363
Punch Driver, 401
PUT, 165, 180, 184
Put Record, 180

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
Index

R

RAD, 109
Radian Mode, 109
RAM, 364
Random Access File, 184
Random Files, 168
Random Number Generator, 201
Randomize, 200
READ, 143
Read Approximate, Primary File (Basic-KSAM), 316
Read Data, 143
Read Current Record, Primary File (Basic-KSAM), 310
Read First Primary Record By Specified Alternate Key (Basic-KSAM), 331
Read Next Primary Record By Current Alternate Key (Basic-KSAM), 332
Read Next Record, Primary File (Basic-KSAM), 311
Read Nth Record, Primary File (Basic-KSAM), 320
Read Previous Record, Primary File (Basic-KSAM), 309
Read Primary Record By Current Alternate Key (Basic-KSAM), 329
Read Random Record, Primary File (Basic-KSAM), 315
Reader Driver, 401
Records, 161
Referencing String Variable, 33
Relational Operator, 42
REM, 95
Remark, 95
Rename (Basic-KSAM Utility), 351
RENAME, 174
Rename File, 174
RENUMBER, 84
Renumber Statement Lines, 84
Reorganize (Basic-KSAM Utility), 350
REPEAT-UNTIL, 131
Repeat-Until Loop, 131
RESTORE, 145
Restore Data Pointer, 145
Retrieve Primary Key, Current Record (Basic-KSAM), 312
RETRY, 121
RETURN, 119
RND, 201
ROM, 364
RUN, 55, 88
Run Program, 88

S

SAVE, 63, 90
SCR, 60
SCRatch, 91
Scratch User Area, 91
Sector, 364
Select Procedure Library, 274
SET, 235

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL

Index

Set System Parameter, 235
Sequential Files, 167
SFMODE, 110
SGN, 202
SHORT, 111
Short Floating Point, 22
Short Floating Point Mode, 110
Short Floating Point Variable, 30
Short Variable, 111
Sign, 202
SIN, 206
Sine, 206
Software, 364
Space, 15, 157
SPC, 157
SQR, 203
Square Root, 203
Statement, 16, 54
Statement Line, 365
Statement Number, 54
Status (Basic-KSAM Utility), 351
STOP, 135
Stop Program Execution, 135
STR\$, 218
String, 23
String Equivalent, 218
String Function, 189
String Internal Machine Representation, 21
String Literal, 27, 365
String Literal Format, 25
String Variable, 32, 54, 366
Symbol [], 353
Symbol < >, 353
Syntax, 15
SYS, 237
System Parameter, 237

T

Tab, 158
TAN, 207
Tangent, 207
THEN, 125, 127
TRACE, 93
Track, 366
Trigonometric Function, 188
TU-ART Driver, 402
TYPE, 248
Type Of Variable, 248

CROMEMCO 32K STRUCTURED BASIC INSTRUCTION MANUAL
Index

U

UNLOCK, 277
Unlock Partition, 277
UNTIL, 131
Update Record, Primary File (Basic-KSAM), 318
Upper Case Character, 15
USE, 275
Use Partition, 275
User Area, 55, 366
User Trappable (Non-Fatal) Errors, 374
USR, 245
Utilities (Basic-KSAM), 346

V

VAL, 219
VALC, 220
Value of String, 219
Value of String With Error Checking, 220
Variable, 51, 52, 366
Variable Representation, 29
Verify Alternate Record (Basic-KSAM), 334

W

While-Endwhile Loop, 132

X

Xor Boolean Operator, 46